An Architectural Approach to Achieving Higher-level Security for Component (Service) Based Software Systems

Artem Vorobiev

Submitted in fulfilment of the requirements of the degree of Doctor of Philosophy

Faculty of Information and Communication Technologies
Swinburne University of Technology

December, 2008
Software systems, in particular component (or service) based software systems, are becoming highly distributed and complex involving independent collaborating components working together towards achieving the systems’ goals. In current practice, a system’s security features are often added after the functional requirements have been addressed. As such, these security features are not systematically designed into the system, and consequently the system often has inherent design flaws and vulnerabilities that can be exploited by intruders, and companies spend much time and resources to fix them up. Meanwhile, the number of security attacks against these systems is also growing. These attacks are more sophisticated and difficult to identify, analyse, correlate (i.e., find out the root attack that triggers other attacks), anti-correlate (i.e., select and enforce proper countermeasures), and mitigate. Therefore, there is a strong need for a systematic software engineering approach, which we call software security engineering (SSE), for developing secure and robust component (or service) based software systems by considering security and functional requirements at the same time.

To address the above issue, we draw on some analogies from the human society and biological systems in which the “strong” can protect the “weak”, the resulting relationship and the whole system that are stronger than the individual “links”. We argue that through collaboration of a system’s constituent components (i.e., distributed detection and defenses) there is a better chance to detect and withstand the new generation of security attacks including multi-phased distributed attacks and various flooding distributed denial of service (DDoS) attacks. Besides, in order to achieve collaborative intrusion detection and defenses in distributed environments, the system and its constituent components should have a mechanism to share with each other a general understanding of information about security attacks and countermeasures. Furthermore, this system should be adaptive and reconfigurable as a measure to withstand security attacks in addition to the traditional approaches such as blocking the IP addresses of the sources of the attack.

Following the above considerations, in this thesis, we introduce a new architectural approach to achieving higher-level security for component (service) based software systems. It includes a reference architecture with defensive components used as a foundation of our approach, a number of security ontologies utilised by different distributed components as a common vocabulary, and a language for describing and manipulating the system design and configurations. First, the reference architecture for managing security called SECROBAT supports defensive components (DCs) including intrusion detection components (IDCs), honey-
pot components (HCs) and key distribution components (KDCs), and adopts the pure peer-to-peer (P2P) and the super-peer (S-P) structures to allow components to operate as a coalition and be adaptive and reconfigurable in order to resist different types of security attacks. Based on SECROBAT different software applications can be developed including collaborative and distributed systems, Web service-based systems, social network systems, and online gaming systems. Second, we develop and apply security ontologies as a common vocabulary for sharing and analysing information among distributed system components such as DCs which collaboratively identify security attacks and realise defensive measures. We adopt an ontological approach because of its flexibility, scalability, reusability, and possibility to evolve over time and solve interoperability problems. Several security ontologies are developed including the security attack ontology (SAO), the security defence ontology (SDO), the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), and the security function ontology (SFO). Third, we design a GIZKA language that is based on SECROBAT and the security ontologies, for specifying dynamic software architectures, their security properties, and security attacks and defenses. It also helps the administrator to manage the system at runtime. GIZKA makes the process of designing, developing, and managing software systems simple and flexible. Finally, our approach is demonstrated through a case study of an example social network system and a prototype implementation.
The Author’s Publications


Acknowledgements

I sincerely thank my supervisor Professor Jun Han. This work could not have been completed without his encouragement, constructive comments and criticism. Professor Han guided me throughout the whole process of this research from inception of the initial idea to the final preparation of this thesis. His broad knowledge and deep insight in software engineering and information security research has helped me many times to take the right directions in the research. I would also like to thank Professor Ryszard Kowalczyk and Dr. Khaled Md. Khan for agreeing to act as associate supervisors.

Thanks to the members of the CeCSES (Component Software and Enterprise Systems research group at Swinburne), in particular, Zheng Li, Dr. Antony Tang, and Dr. Yan Jin for their feedback, help, suggestions, friendship and encouragement. Acknowledgement is also due to the members of IBM Tokyo Research Lab and the many anonymous reviewers of my conference papers on which this thesis was based, and whose comments helped to tune this work. A special thank goes to Professor Elena Ivanova for introducing me to the area of Microbiology in this research.

Finally, I would like to express my profound gratitude to my family, Nargiza, Galina, Alexander, and Evgeny, for their patience and support over past four years.
Declaration

This thesis contains no material which has been accepted for the award to the candidate of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of the candidate’s knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Artem Vorobiev

December, 2008
Table of Contents

Abstract.............................................................................................................................................. i
The Author's Publications ................................................................................................................ iii
Acknowledgements ........................................................................................................................... iv
Declaration........................................................................................................................................ v
Table of Contents ........................................................................................................................... vi
List of Figures ................................................................................................................................... xii
List of Tables .................................................................................................................................... xvii

Chapter 1: Introduction ..................................................................................................................... 1
  1.1 Security of component based software systems .................................................................. 2
    1.1.1 Component based software systems .......................................................................... 2
    1.1.2 Security ....................................................................................................................... 2
  1.2 Research aims and objectives................................................................................................. 3
  1.3 Overview of our approach and contributions ...................................................................... 4
  1.4 Structure of the thesis .......................................................................................................... 6

Chapter 2: Related work .................................................................................................................... 8
  2.1 Information security .............................................................................................................. 8
    2.1.1 Cryptography and cryptanalysis ................................................................................ 9
    2.1.2 Information hiding ................................................................................................... 12
    2.1.3 Methods of identification and authentication ......................................................... 13
    2.1.4 Operating systems security ...................................................................................... 13
    2.1.5 Network security ..................................................................................................... 15
    2.1.6 Security evaluation standards .................................................................................. 16
    2.1.7 Summary ................................................................................................................. 17
  2.2 Security in distributed software systems ............................................................................. 17
    2.2.1 Object-oriented middleware platforms .................................................................... 17
    2.2.2 Security in peer-to-peer systems ............................................................................. 24
    2.2.3 Web services and their security standards ............................................................... 25
    2.2.4 Summary ................................................................................................................. 27
  2.3 The art of creating secure architectures................................................................................. 28
    2.3.1 Introduction ............................................................................................................. 29
    2.3.2 Security requirements .............................................................................................. 30
    2.3.3 Analysis of software architectures and security protocols .................................... 31
    2.3.4 Design techniques .................................................................................................. 32
    2.3.5 Tools and languages ............................................................................................... 33
    2.3.6 Security testing ....................................................................................................... 34
    2.3.7 Summary ................................................................................................................. 35
  2.4 Summary ............................................................................................................................... 35
Chapter 5: Security attack and defence ontologies ................................................................. 93
  5.1 SAO and SDO classes ...................................................................................................... 93
  5.2 The classes WSAAttack and WSDefence ................................................................. 94
    5.2.1 Web service attacks and defenses ................................................................. 94
    5.2.2 The class DiscoveryAttack .............................................................................. 96
    5.2.3 The class XMLAttack ...................................................................................... 97
    5.2.4 The class WSDoSAttack ............................................................................... 99
    5.2.5 The class ApplicationAttack ......................................................................... 100
    5.2.6 The class SOAPAttack ............................................................................... 101
    5.2.7 The class SemanticAttack ............................................................................. 102
    5.2.8 The class WSDefence .................................................................................... 102
  5.3 Classes P2PAttack and P2PDefence ............................................................................... 106
    5.3.1 Attacks against P2P and countermeasures .................................................... 106
    5.3.2 Classes P2PInfrastructureAttack & P2PInfrastructureDefence .................... 107
    5.3.3 Classes P2PSemanticAttack and P2PSemanticDefence .................................. 108
    5.3.4 Classes P2PNetworkAttack and P2PNetworkDefence .................................. 110
    5.3.5 Classes P2PAnonymityAttack and P2PAnonymityDefence ......................... 111
  5.4 DoS attacks and countermeasures .................................................................................. 111
    5.4.1 The class DoSAttack ..................................................................................... 112
    5.4.2 The class DDoSAttack ............................................................................... 113
    5.4.3 The class DRDoSAttack ............................................................................... 113
    5.4.4 The class DoSDefence ................................................................................... 114
    5.4.5 The class DDoSDefence ............................................................................... 114
    5.4.6 The class DRDoSDefence ............................................................................. 115
  5.5 Classes SniffingAttack and SniffingDefence ................................................................. 116
  5.6 Classes of multi-phased distributed attacks and defenses ........................................... 117
    5.6.1 The traditional Mitnick attack ...................................................................... 118
  5.7 Rules .......................................................................................................................... 119
    5.7.1 Alert processing ............................................................................................. 119
  5.8 Example: WS Mitnick attack ....................................................................................... 122
    5.8.1 Defence against multi-phased distributed attacks ......................................... 124
  5.9 Summary ................................................................................................................... 126

Chapter 6: GIZKA: a language for managing security .......................................................... 128
  6.1 Introduction to GIZKA ............................................................................................... 129
  6.2 Specification of security properties ............................................................................... 131
    6.2.1 Motivation example ..................................................................................... 131
    6.2.2 Introduction to SCL .................................................................................... 133
    6.2.3 Extended SCL (GIZKA.ESCL) ...................................................................... 136
    6.2.4 Additional constructs for specifying security properties ............................... 142
    6.2.5 Reasoning ..................................................................................................... 145
  6.3 Description of attack correlation and anti-correlation ................................................... 149
    6.3.1 Introduction to attack languages .................................................................... 150
    6.3.2 Attack identification ...................................................................................... 151
Chapter 7: A social network system as a case study

7.1 Business plan .......................................................... 176
  7.1.1 Social network system ............................................ 176
  7.1.2 Market ............................................................. 177
  7.1.3 Industry analysis ................................................ 178

7.2 Requirements ............................................................. 179
  7.2.1 User’s requirements ............................................. 179
  7.2.2 Provider’s requirements ...................................... 180

7.3 SNS and specification of its architecture ......................... 180
  7.3.1 Overview and scenarios ....................................... 181
  7.3.2 Style PeerStyle .................................................... 184
  7.3.3 Specification of the SNS ....................................... 185

7.4 Security properties ..................................................... 187
  7.4.1 Distribution of shared keys ................................... 187
  7.4.2 Storing data ....................................................... 191
  7.4.3 Secure connections ............................................. 193

7.5 Dealing with security attacks ....................................... 194
  7.5.1 Dealing with Mitnick attacks ................................. 194
  7.5.2 Dealing with flooding DoS attacks through structural reconfiguration ............. 199

7.6 Summary ........................................................................ 201

Chapter 8: A prototype of a social network system ................. 202

8.1 General overview ........................................................ 202

8.2 Peer architecture ........................................................ 204
  8.2.1 The microkernel .................................................. 205
  8.2.2 PROTO modules ................................................. 205

8.3 Peers ............................................................................ 206
  8.3.1 Defensive peers ..................................................... 206
  8.3.2 Manager peers ..................................................... 207
  8.3.3 Reserved peers ..................................................... 208
  8.3.4 Storage server peers ............................................ 209
  8.3.5 Client peers ........................................................ 209

8.4 Specifying security properties ....................................... 209
8.4.1 Reasoners of probability, time and security properties ............................................. 209
8.5 Security attacks and defenses ....................................................................................... 212
8.5.1 Development tools used to specify security ontologies ............................................. 212
8.5.2 Reasoning about attacks and countermeasures ......................................................... 215
8.6 Deployment .................................................................................................................... 217
8.7 Simulation ...................................................................................................................... 218
8.8 Limitations .................................................................................................................... 219
8.9 Summary ....................................................................................................................... 221

Chapter 9: Conclusions and further research ................................................................ 222
9.1 Summary ....................................................................................................................... 222
9.2 Key contributions ......................................................................................................... 223
9.2.1 SECROBAT: a reference architecture ................................................................. 224
9.2.2 Security ontologies ................................................................................................. 224
9.2.3 GIZKA: a language for managing security .............................................................. 225
9.2.4 Case study and prototype ....................................................................................... 226
9.2.5 Conclusion ............................................................................................................... 227
9.3 Further research ......................................................................................................... 227

References .......................................................................................................................... 228

Appendix A ............................................................................................................................. 239
A.1 SecurityAlgorithm: symmetric cryptography algorithms ............................................... 239
A.2 SecurityConcept: secure communications ................................................................. 240
A.3 Security functions ........................................................................................................... 242
A.3.1 Security functions for cryptographic support ......................................................... 242
A.3.2 Security functions for user data protection ............................................................ 243
A.3.3 Security functions for identification and authorisation ........................................... 243
A.3.4 Security functions for security audit and resource utilisation .................................. 244
A.3.5 Security functions for privacy .............................................................................. 244
A.3.6 Security functions for trusted channels ................................................................. 244

Appendix B ............................................................................................................................. 245
B.1 P2P attacks and defences .............................................................................................. 245
B.1.1 Classes P2PInfrastructureAttack and P2PInfrastructureDefence .......................... 245
B.1.2 Classes P2PNetworkAttack and P2PNetworkDefence ............................................. 246
B.1.3 Classes P2PAnonymityAttack and P2PAnonymityDefence .................................... 247
B.2 DDoS attacks and defences .......................................................................................... 249
B.2.1 DDoS attacks ......................................................................................................... 249
B.2.2 DDoS defence ....................................................................................................... 251

Appendix C ............................................................................................................................. 253
C.1 Message layer ............................................................................................................... 253
C.1.1 MessageInspector .................................................................................................. 253
List of Figures

Figure 1.1. The high-level picture of our research approach .................................................... 4
Figure 2.1. Object request broker architecture [SGM02] ........................................................... 18
Figure 2.2. CORBA security model [DBD+03] ........................................................................ 19
Figure 2.3. .NET framework [NET05a] .................................................................................. 20
Figure 2.4. .NET security structural model [MSD06] ............................................................... 21
Figure 2.5. Java 2 Platform Standard Edition 5.0 [Java06] ..................................................... 22
Figure 2.6. J2EE Platform APIs 1.4 [Java06] ......................................................................... 23
Figure 2.7. The Java 2 Platform - Compile and Execute [DBD+03] ......................................... 23
Figure 2.8. The general process of engaging Web services [BHM+04] ................................. 26
Figure 2.9. Web services security specifications [IM02] ......................................................... 27
Figure 2.10. SDLC [BM05] .................................................................................................... 29
Figure 3.1. Example system .................................................................................................. 41
Figure 3.2. Model for managing security ............................................................................... 41
Figure 3.3. Centralized P2P .................................................................................................. 44
Figure 3.4. Pure P2P ............................................................................................................ 44
Figure 3.5. Hybrid P2P ......................................................................................................... 45
Figure 3.6. P2P architecture .................................................................................................. 45
Figure 3.7. The structure ...................................................................................................... 48
Figure 3.8. The architecture in a normal regime ..................................................................... 51
Figure 3.9. The architecture during attacks .......................................................................... 51
Figure 3.10. The SECROBAT-based gaming system in the normal regime ....................... 56
Figure 3.11. The SECROBAT-based gaming system during attacks .................................... 56
Figure 4.1. The security asset-vulnerability ontology ............................................................. 62
Figure 4.2. The class ‘Consequence’ ..................................................................................... 64
Figure 4.3. The class ‘Asset’ ................................................................................................ 64
Figure 4.4. The class ‘AssetValue’ ....................................................................................... 64
Figure 4.5. The class ‘Vulnerability’ ..................................................................................... 65
Figure 4.6. Interrelations among SASO and SFO ................................................................. 66
Figure 4.7. The class ‘SecurityObjective’ ............................................................................. 67
Figure 4.8. The class ‘SecurityAlgorithm’ ............................................................................ 68
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.20</td>
<td>MessageLayer</td>
<td>173</td>
</tr>
<tr>
<td>6.21</td>
<td>Relationships among security patterns</td>
<td>174</td>
</tr>
<tr>
<td>7.1</td>
<td>SNS in the normal regime</td>
<td>182</td>
</tr>
<tr>
<td>7.2</td>
<td>SNS during attacks</td>
<td>183</td>
</tr>
<tr>
<td>7.3</td>
<td>The style ‘PeerStyle’</td>
<td>184</td>
</tr>
<tr>
<td>7.4</td>
<td>Specification of the SNS</td>
<td>187</td>
</tr>
<tr>
<td>7.5</td>
<td>Security properties of KDC1 for IDC1 and IDC2</td>
<td>188</td>
</tr>
<tr>
<td>7.6</td>
<td>Security properties of IDC1 for IDC2 and KDC1</td>
<td>189</td>
</tr>
<tr>
<td>7.7</td>
<td>Security properties of IDC2 for IDC1 and KDC1</td>
<td>190</td>
</tr>
<tr>
<td>7.8</td>
<td>Security properties of SS1 for ‘storeData()’ functionality</td>
<td>192</td>
</tr>
<tr>
<td>7.9</td>
<td>Security properties of Client1 for ‘storeData()’ functionality</td>
<td>192</td>
</tr>
<tr>
<td>7.10</td>
<td>Security properties of IDC1 for ‘organiseConnection()’ functionality</td>
<td>193</td>
</tr>
<tr>
<td>7.11</td>
<td>Security properties of IDC2 for ‘organiseConnection()’ functionality</td>
<td>193</td>
</tr>
<tr>
<td>7.12</td>
<td>The Mitnick attacks against DC2</td>
<td>195</td>
</tr>
<tr>
<td>7.13</td>
<td>Specification of SNS</td>
<td>200</td>
</tr>
<tr>
<td>8.1</td>
<td>General overview</td>
<td>203</td>
</tr>
<tr>
<td>8.2</td>
<td>Peer architecture</td>
<td>204</td>
</tr>
<tr>
<td>8.3</td>
<td>Defensive peer</td>
<td>206</td>
</tr>
<tr>
<td>8.4</td>
<td>Manager peer</td>
<td>208</td>
</tr>
<tr>
<td>8.5</td>
<td>The architecture of the reasoning engine</td>
<td>210</td>
</tr>
<tr>
<td>8.6</td>
<td>Specifying the first five classes of SDO</td>
<td>213</td>
</tr>
<tr>
<td>8.7</td>
<td>The final result</td>
<td>213</td>
</tr>
<tr>
<td>8.8</td>
<td>Depicting SDO through OWLViz</td>
<td>214</td>
</tr>
<tr>
<td>8.9</td>
<td>Jambalaya nesting view</td>
<td>214</td>
</tr>
<tr>
<td>8.10</td>
<td>The class ‘BandwidthDepletionAttack’</td>
<td>215</td>
</tr>
<tr>
<td>8.11</td>
<td>BASE</td>
<td>216</td>
</tr>
<tr>
<td>8.12</td>
<td>Backtrack 2</td>
<td>219</td>
</tr>
<tr>
<td>C.1</td>
<td>MessageInspector</td>
<td>254</td>
</tr>
<tr>
<td>C.2</td>
<td>MessageInterceptorGateway</td>
<td>254</td>
</tr>
<tr>
<td>C.3</td>
<td>SingleAccessCheckPoint</td>
<td>255</td>
</tr>
<tr>
<td>D.1</td>
<td>Example modules</td>
<td>262</td>
</tr>
<tr>
<td>D.2</td>
<td>Methods ‘handleMessage()’ and ‘getAcceptedTypes()’</td>
<td>265</td>
</tr>
<tr>
<td>D.3</td>
<td>The class ‘StorageTCPSendableMessage’</td>
<td>266</td>
</tr>
<tr>
<td>D.4</td>
<td>The class ‘StorageServer’</td>
<td>266</td>
</tr>
</tbody>
</table>
Figure D.5. The Eclipse SDK ................................................................................................ 267
Figure D.6. The class Database ............................................................................................ 268
Figure D.7. The dialog between two clients .......................................................................... 269
Figure D.8. The dialog (1) .................................................................................................. 269
Figure D.9. The dialog (2) .................................................................................................. 270
Figure D.10. Alert processing ............................................................................................... 272
Figure D.11. Snort alerts ..................................................................................................... 273
List of Tables

Table 4.1. Security functions for cryptographic support ....................................................... 81
Table 4.2. Security functions for user data protection ........................................................... 83
Table 4.3. Security functions for identification & authorisation ........................................... 84
Table 4.4. Security functions for security audit and resource utilisation .............................. 85
Table 4.5. Security functions for privacy .............................................................................. 86
Table 4.6. Security functions for trusted channel ................................................................. 86
Chapter 1

Introduction

Software systems have become inexorably more open, complex, distributed, pervasive and mobile, especially component (or service) based software systems, engaging many various components working together to achieve the systems’ goals. This accelerating trend is driven by new technologies that take advantage of the increasing power and failing cost of hardware and network resources. Such systems need to interact with other systems creating increasingly heterogeneous and dynamic environments. Besides, a system’s security features are usually added after functional requirements have been addressed, i.e., these features are not systematically designed into the system, and consequently the system has inherent security “holes”. Such software systems may have a lot of vulnerabilities that may be exploited by attackers, and companies spend much time and money to “patch” them up. At the same time, a new generation of security attacks (e.g., multi-phased distributed attacks or various flooding distributed denial of service attacks) appear that are difficult to identify, analyse, correlate (i.e., detect the triggering root attack), anti-correlate (i.e., choose and deploy a proper countermeasure), and mitigate.

Therefore, there is a strong need for a systematic engineering approach to creating secure and robust component (service) based software systems by considering functional and security requirements at the same time. Moreover, through collaboration of system’s constituent components used for distributed detection and defenses it may be possible to identify and withstand security attacks. Furthermore, these components can be organised in such an effective way using security properties of each other to allow the system to adapt and reconfigure its structure dynamically in order to resist security attacks. However, to communicate to each other effectively and in a trusted way, these components need a common vocabulary to share a common understanding of security related information, especially about security attacks and countermeasures, among each other. In particular, various distributed constituent components of these systems should securely and in a trusted manner communicate to each other regarding security incidents through the use of a common vocabulary compressible to both humans and software agents in order to effectively detect, analyse, and mitigate effects of various distributed security attacks. Moreover, such systems should support various security mechanisms and be adaptive and reconfigurable in order to withstand different types of security attacks.
1.1 Security of component based software systems

In this section, we introduce basics of component based software systems and their security.

1.1.1 Component based software systems

In component based software systems (CBSSs), a composition may be static or dynamic and include various distributed over the network components which have their own data and executable code. In static composition, two components interact to each other using a connector, while in a dynamic composition, two components communicate through ports and communication networks. There are two types of components, distributed and downloadable. Distributed components are used in dynamic compositions and run on remote servers while downloadable components are employed in static compositions and executed locally. Besides, components can be atomic (primitive system blocks) or composite (consist of atomic components). Two software components bind to each other through the use of their interfaces. A composition of atomic and composite components results in a software architecture. Information is usually transferred among components through connectors represented by events, pipes, procedure calls, and some others. The interface signature of the component specifies the exchange protocols among connectors. It presents the data and control data such as attributes and operation parameters. Even though software components are “black-boxes”, they can be used in compositions because their functionalities are specified through the interfaces. The examples of standard interface signatures include Application Programmer Interface (API), the Java Database Connectivity (JDBC) [Java06] and Object Adaptors in CORBA [CORBA07].

1.1.2 Security

Information security applies to all aspects of protecting information in any form. Actually, true information security is not possible [STC05, Wiki07] and should be treated as a continues process. Currently, information security is moving beyond the perimeter and becoming more data-focused, protecting data both at rest and in transit. In addition, information security is a multi-dimensional concept that includes such high level objectives as authentication, access control, auditability, confidentiality, integrity, availability, and non-repudiation [Dos01], and some others. The detailed description of the field of information security is given in the next chapter.

Software component security focuses on the protection of components’ resources and data, and various properties of components and a whole CBSS as well. It is necessary to mention that security of components depends on their functional role and place in the system. In addition, software components are usually black boxes (in binary form). Due to unavailability to check the source code of components and verify their security properties, software integrators cannot be sure about security of the final composition. Besides, CBSSs usually include various
distributed over the network components. A component may be proved to be secure in one composition, however, it may be insecure in another one. Component security can be based on various relative elements (tend to change continuously) such as the value of data or the use domain. Because of the components being black boxes integrators cannot base their assumptions on just components’ claims about their security or insecurity. Hence, integrators should have control over the security properties of individual components. Current approaches to component security mainly focus on adding more security functions such as encryption, hash functions etc. These approaches are important but do not guarantee component security. Hence, other approaches should be applied which will allow to evaluate and analyse the security properties of the components in order to make a secure composition. Components assembled securely in one composition may not guarantee security of another composition. Each component should provide the description of its required and ensured security properties to allow software integrators to reason about their security properties and predict security of the whole CBSS.

Also, because of the distributed nature of CBSSs it experience a new generation of distributed security attacks which are difficult to identify and mitigate. Since these attacks may be composite, i.e., multi-staged and distributed and include various simple attacks, it can be a very difficult task to detect and resist them. We argue that the components should act as a coalition and share information about attacks in a form comprehensible to humans and software agents. Besides, there is no the silver bullet solution for another problem such as the flooding distributed denial of service (DDoS) attacks. The common practice is to block the source of the security attack. However, by doing this the owner of a CBSS also may block many legitimate users who belong to the same segment of the networks as the blocked malicious machines. To solve this issue we propose the approach which allows a CBSS to reconfigure and adapt dynamically its structure in order to resist attacks.

1.2 Research aims and objectives
The aim of this research is to develop an architectural approach to achieving higher-level security for component (service) based software systems by considering security and functional requirements at the same time. This approach is driven by the analogies from the human society and biological systems in which the “strong” can protect the “weak” where the resulting relationship and the whole system become even stronger than the strongest link (or individual). This approach argues that through collaboration of a system’s constituent components there is a better chance to register, analyse, and resist the new generation of security attacks including multi-phased distributed and DDoS attacks. Such collaboration is controlled through the use of components’ security properties in order to allow software components to interact with each other and verify the compatibility of their security properties and build a secure CBSS. In
addition, this approach creates a mechanism represented by a common vocabulary that lets collaborating components share with each other a common understanding of information about security attacks and defenses in order to achieve collaborative attack detection and defenses in dynamic distributed environments. Moreover, this research aims to develop the system with be adaptive and reconfigurable structure in order to resist and survive during security attacks.

The key objective of this research is to investigate and develop an architectural approach to achieving higher-level security for component (service) based software systems. In contrast, the broad objectives are formulated as:

- **Objective 1:** To create a reference architecture with defensive components and an adaptable and reconfigurable network structure and employ it as a basis for developing component (service) based software systems capable of identifying distributed security attacks, analysing and resisting them.

- **Objective 2:** To develop a vocabulary comprehensible to humans and software agents in order to support Objective 1 and allow system’s distributed components to share information regarding security attacks and defenses in a trusted way. Besides, a vocabulary should be used for identifying, analysing, correlating, and anti-correlating these security attacks.

- **Objective 3:** To let components capture and analyse security properties of other components with which they interact and verify the compatibility of their security properties.

- **Objective 4:** To create a language that formally defines Objectives 1, 2, and 3 and makes the process of designing, developing and managing of software systems based on our approach as simple and flexible as possible.

- **Objective 5:** To produce a case study and build a generic system prototype based on the outcomes of Objectives 1, 2, 3, and 4.

### 1.3 Overview of our approach and contributions

![Figure 1.1. The high-level picture of our research approach](image-url)
Our research approach is schematically illustrated in Figure 1.1 and consists of several contributions.

We develop a reference architecture, called SECROBAT, with defensive components (intrusion detection components, honey-pot components, and key distribution components) and the hybrid P2P network structure. This reference architecture helps us to design and develop the secure and robust component-based software system and drives the whole approach while biological systems serve as an inspiration.

Because of the distributed nature of the reference architecture, its different components need to communicate and warn each other regarding security attacks or environment changes. Hence, we adopt an ontological approach because we need to define our vocabulary in a way understandable to humans and software agents. Our security ontologies are flexible, scalable, reusable, can evolve over time, solve interoperability problems and can be used by different distributed components, especially defensive components, in order to detect various security attacks, correlate and anti-correlate them, and develop and deploy countermeasures. We create several security ontologies including the security attack (SAO) and defence ontologies (SDO), the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), and the security function ontology (SFO).

We develop a language for managing security called GIZKA that allows to specify component (or service) based software systems in a formal manner. Besides, GIZKA.ATTACK (a subset of GIZKA) is capable of defining security attacks in a formal way in order to detect possibly interrelated single attacks, analyse them and find out the root attack that triggers them, and select a proper countermeasure against a particular attack (anti-correlation). Further, we create an architectural description language (a part of GIZKA) called GIZKA.ADL which is based on ACME/Armani and suitable for describing dynamic architectures, their structural reconfigurations and adaptation mechanisms, and especially their security properties. Moreover, we have applied and extended the Security Characterization Language (SCL) called GIZKA.ESCL suitable for specifying security interfaces of software components (or services) and security contracts that govern connections among different components using our reasoning rules. As a countermeasure against distributed security attacks (e.g., Smurf or Fraggle attacks) targeting the system, we adopt structural reconfiguration and adaptation of the system. Therefore, GIZKA is used to describe and analyse dynamic reconfigurations at design time as well as at runtime. Moreover, in order to manage a system’s security, GIZKA allows to control and deploy dynamic reconfigurations using security patterns and the security pattern ontology (SPO).

We demonstrate the applicability of our approach to the real world through the use of a social network system and its business plan served as a case study.
Finally, a prototype is developed based on SECROBAT, the security ontologies, and GIZKA. The prototype has a microkernel architecture consisted of modules connected to the microkernel. Such approach helps us to create components and the whole system in a very fast, robust, and secure way. The main advantages of the prototype are an implemented secure protocol, a microkernel architecture, a specification (internal language) for developing secure software modules and applications, a secure messaging mechanism, a method for developing software modules, a large collection implemented software modules, and so on. To be as “light” and simple as possible and cross-platform at the same time, we implement the prototype using Java and Groovy utilised as a script language.

1.4 Structure of the thesis

The rest of the thesis is organised as follows:

- In this chapter (Chapter 1), we have presented an introduction, a problem statement, objectives, and an overview of our approach. In other words, this chapter has provided a foundation and context for the following chapters of the thesis setting the scene for the detail discussion of the main research concepts presented in the subsequent chapters.

- In Chapter 2, we introduce a literature review summarizing current techniques employed to design and develop software systems. We report about several subsets of information security and highlight some important issues. Besides, we describe several distributed technologies, compare them, and analyse their security features. We present an area called Software Security Engineering (SSE), and then describe and analyse different approaches for developing and evaluating secure software systems. Finally, we highlight several open issues for further research.

- In Chapter 3, with biological systems as an inspiration, we introduce a reference architecture, called SECROBAT, utilised for developing component-based (or service-based) software systems. We describe the need for this reference architecture and how it can be implemented. Moreover, we present its key features including defensive components and the hybrid P2P network structure. We explain how various contributions of our research work including the security ontologies and the GIZKA language relate to each other and are used as a whole model. Finally, we demonstrate our approach through the use of the gaming system.

- In Chapter 4 and Chapter 5, we present the security ontologies utilised as a vocabulary to share a common understanding about different security incidents among humans and software agents, describe the need for them and their usage, and explain reasons of selecting them over other classifications. We provide a detailed description of the security asset-vulnerability ontology, the security algorithm-standard ontology, the security function
ontology, the security attack ontology, and the security defence ontology. Then, we illustrate the use of these security ontologies using an example of a Mitnick attack.

- In Chapter 6, we describe our language, called GIZKA, that allows to specify dynamic security properties and reasoning rules, define and analyse dynamic structural reconfigurations, describe security attacks in a formal way in order to analyse them and consequently correlate and anti-correlate. As an example of a countermeasure against multi-phased distribute attacks and distributed denial of service attacks, we present dynamic structural reconfiguration and adaptation mechanisms that use security patterns and various security techniques.

- In Chapter 7, we introduce a case study (i.e., a social network system) and use it to demonstrate our approach.

- In Chapter 8, we describe the prototype implementation of our framework to illustrate how the research outcomes of this thesis could be implemented and utilised.

- Finally in Chapter 9, we summarise our research work and highlight our major contributions and point to further research.
Chapter 2

Related work

Modern software systems are becoming highly distributed and complex. Moreover, a system’s security features are not systematically designed into the system because they are usually added after addressing all functional requirements. Consequently, the system inherits various design flaws and vulnerabilities which may be used by intruders and companies spend much time and money to fix them up. At the same time, the number of security attacks, which are more sophisticated and difficult to identify, analyse, and mitigate, against these systems is also growing. Therefore, there is a strong need for a systematic software engineering approach to developing secure software systems.

In this chapter, we review current methods and techniques that are used to design and implement security into software systems, and analyse them from a software engineering perspective. First, we review several subsets of information security techniques including cryptography, information hiding, passwords, network and operating systems security, and the Common Criteria. Then, we introduce, compare and analyse in terms of their security features, several distributed technologies including the Common Object Request Broker Architecture (CORBA), .NET, Java 2 Enterprise Edition (J2EE), Peer-to-Peer (P2P), and Web services. Next, we present Software Security Engineering (SSE), describe and analyse a number of approaches to creating and evaluating secure software applications. Furthermore, we review a number of dynamic software architectures and approaches used to specify them. Finally, we draw some conclusions and summarise important issues for further investigation. It is worth to note that every chapter of this thesis contains some further literature review, hence, the research works introduced in this chapter cover only some of the broad topics related to this approach.

2.1 Information security

Information security applies to all aspects of safeguarding or protecting information/data in any form. Computer (network) security, information technology (IT) security, information systems security, information and communications technology (ICT) security, and even physical security are all subsets of information security [STC05,Wiki07]. On the other hand, the International Information Systems Security Certifications Consortium (ISC)2 [ISC2] defines ten Common Body of Knowledge (CBK) domains for the Certified Information Systems Security Professional (CISSP) certification:
• Access Control;
• Application Security;
• Business Continuity and Disaster Recovery Planning;
• Cryptography;
• Information Security and Risk Management;
• Legal, Regulations, Compliance and Investigations;
• Operations Security;
• Physical (Environmental) Security;
• Security Architecture and Design;
• Telecommunications and Network Security.

Furthermore, information security of software systems is a multi-dimensional concept that includes authentication, access control, auditability, confidentiality, integrity, availability, and non-repudiation [Dos01]. Information security can be specified through the use of “CIA triad” that includes three principal classes of security objectives: confidentiality (concerned with protecting data from unauthorised use or users), integrity (concerned with maintaining the consistency and correctness of information), and availability (preventing the exhaustion of resources and the denial of services to authorised users). Also, it can be described as “AAA” which stands for “authentication (validating who the entity claims to be), authorisation (related to the access to system resources such as data and operations), and accountability (subjects have to be accountable for their actions)”. “AAA” becomes “AAAA” if is combined with auditability (detecting, recognising, recording, analysing security related activities). As it can be seen, information security is a rather complex research area without the certain definition the term “information security” [STC05, Wiki07]. It is worth mentioning that in addition to “CIA triad” and “AAAA” we utilise several other security objectives described by non-repudiation (preventing an individual or entity from denying having performed an action such as sending or receiving information), identification (the process by which a subject provides an identity), and survivability which deals with functioning during and after attacks.

In this section, we describe a few subsets of information security including cryptography, information hiding, passwords, network and operating systems security, and security systems evaluation standards such as the Common Criteria. Besides, we highlight a few important open issues in these areas.

### 2.1.1 Cryptography and cryptanalysis

Cryptography is the art of keeping messages secret while cryptanalysis is the art of breaking ciphers [Cry07, RSA07, STC05]. In cryptography, encryption is the process of transforming information (plaintext) through the use of an algorithm (cipher) to make it unreadable to anyone
except those having special knowledge such as a key. This process results in encrypted information called ciphertext. Decryption is the reverse process which makes the encrypted information readable (unencrypted). It should be mentioned that encryption only provides confidentiality of information, hence, other techniques are required to protect integrity and authenticity (e.g., message authentication code (MAC) and digital signatures) as described below. We introduce several subsets of cryptography here while the detailed description is presented in Chapter 4.

**Block Ciphers and stream ciphers**

A cipher is an algorithm for performing encryption and decryption. It is necessary to mention that development of new ciphers is a rather difficult task. There are two types of ciphers, block and stream. Stream ciphers (symmetric encryption algorithms) encrypt and decrypt one small (bit or byte) unit of data at a time while block ciphers encrypt large chunks of data at once. Stream ciphers can be much faster than any block cipher. However, they are vulnerable to attack if certain precautions such as 1) keys should not be used twice and 2) valid encryption should not be relied on to indicate authenticity, are not followed. The encryption of any plaintext with a block cipher results in same ciphertext if the same key is utilized. But with a stream cipher, the transformation of these smaller plaintext units varies. A stream cipher generates a keystream (a sequence of bits used as a key). Encryption is accomplished by combining the keystream with the plaintext (the bitwise XOR operation). The generation of the keystream can be independent of the plaintext and ciphertext (a synchronous stream cipher) or it can depend on the data and its encryption (a self-synchronizing stream cipher). The main goals of block ciphers are diffusion (small change in plaintext, changes lots of cipher text and statistical properties of plaintext hidden in ciphertext) and confusion (statistical relationship between key and ciphertext should be as complex as possible). Therefore developers have to design functions that produce diffused and confused output. The examples of block ciphers are Fiestel cipher, Data Encryption Standard (DES), Triple-DES, and AES. However, there is no stream cipher that has emerged as a de facto standard. The examples of stream cipher are RC4, DES in CFB and OFB modes. Yet, any block cipher can be used as a stream cipher. It is worth mentioning that block ciphers are vulnerable to many cryptographic attacks such as crypto analytical or statistical attacks and many others [STC05,Wiki07].

**Public and private key cryptography**

Public-key cryptosystems are responsible for distributing cryptographic keys and allow people to communicate securely without having to meet first and establish a key these tasks. Public-key (asymmetric) cryptography is a subset of cryptography in which a user has a pair of
cryptographic keys (public/private keys). The private key must be kept secret, while the public key is widely shared. Both keys are related mathematically, however, a message that is encrypted with the public key cannot be decrypted without the corresponding private key. On the other hand, secret key (symmetric) cryptography uses a single secret (private) key for both encryption and decryption. The two main subsets of public key cryptography are 1) public key encryption where a message is encrypted with a recipient's public key and can be decrypted only with the recipient’s corresponding private key to ensure confidentiality and 2) digital signatures where a message, which is signed with a sender's private key, can be verified by anyone who has the sender's public key to ensure authenticity. Rivest-Shamir-Adelman (RSA) Encryption is an example of the public key cryptosystem. However, the main problem for public-key cryptography is proving that a public key is authentic. The usual approach is to use a public-key infrastructure (PKI) where certificate authorities certify digital certificates that contain information regarding ownership of key pairs. PGP (Pretty Good Privacy) uses another approach called the "web of trust" to ensure authenticity of key pairs.

Private-key algorithms use the same key for both encryption and decryption. Diffie-Hellman (D-H) Key Agreement is the example of such algorithm, which is widely used in SSL, Cisco encrypting routers, etc.

However, all public-key (and private-key) schemes are susceptible to brute force key attacks, mathematical attacks, and timing attacks, man in the middle attacks, and others presented in [Wiki07].

(Digital signatures)
As previously mentioned, digital signatures provide an ability to verify author’s authenticity and can be checked by third parties to resolve disputes. However, the human error factor is one of the most common security problems because private keys can be stolen or someone may type a word incorrectly. Also, there are issues regarding storage and backup since digital signature passwords, private keys, and certificates have to be stored somewhere and this gives intruders possibilities for unauthorized or illegal access to passwords and key data. Other issues are the loss of data and the break of cryptographic codes.

(Message authentication)
Message authentication concerns with protecting of message integrity, validating identity of originator, and non-repudiation. For these purposes, three functions are employed including 1) message encryption, 2) hash functions such as MD2, MD4, MD5, and Secure Hash Algorithm (SHA-1 and SHA-256), and 3) message authentication code (MAC) (a cryptographic checksum). However, MD2, MD4, MD5, and SHA-1 are not considered secure anymore.
Other types of cryptography

(Elliptic curve cryptography)

Majority of public-key cryptosystems such as RSA or D-H use either integer or polynomial arithmetic with very large numbers/polynomials which impose a huge load on storing and processing of keys and messages. However, one of the promising alternatives is elliptic curve cryptography (ECC) which offers same security but supports smaller bit sizes (about 1:7).

(Visual cryptography)

Visual Cryptography (VC) is rooted in secret sharing and encryption/decryption of images. Images are split into two shares that, when superimposed, reveal images. Because the decoding process in VC is performed by human eyes, it has such problems as pixel expansion and the loss of contrast.

(Quantum cryptography)

Quantum cryptography (QC) is an approach to securing communications based on certain phenomena of quantum physics. Instead of using mathematical algorithms, QC utilizes physical means such as photons or electrons which restrict attackers from learning the contents of encrypted messages. However, QC is rather difficult to implement. It is applied for creating Quantum Key Distribution (QKD) algorithms or Quantum Cash.

2.1.2 Information hiding

Information hiding consists of two parts including steganography [Joh07] and watermarking [Kal04]. There are three aspects in information-hiding systems: capacity (the amount of information that can be hidden), security (attacker’s inability to detect hidden information), and robustness (the amount of modification the stego medium can withstand before an adversary can destroy hidden information) [PH03]. Steganography is the art of concealing the existence of information within seemingly inoffensive carriers. Its primary goals include high security and capacity. Watermarking is the art of modifying audio-visual and text content in such way that the modifications carry retrievable information and are imperceptible at the same time. In addition, the primary goal of watermarking is to achieve a high level of robustness and impossibility to remove a watermark without degrading the quality of the object. In addition, content has to be difficult to remove and change by unauthorized parties. Attacks on watermarking include simple waveform processing (compression, linear filtering, additive noise, and quantization), detection-disabling methods (geometric transformations), advanced jamming/removal (intentional processing to impair/defeat watermark), and ambiguity/deadlock issues such as reducing confidence in watermark integrity.
2.1.3 Methods of identification and authentication

Traditionally, there are three main and two additional methods that allow to identify and authenticate a user including what a user knows (password or pass phrase), what a user has (a smartcard or electronic token), who a user is (biometrics), and additionally, where a user is (IP or physical address) and what a user is doing (signing an electronic document).

Passwords are employed to protect any kind of data from unauthorized users and usually used in conjunction with logins. However, there are several authentication problems:

- Users need to store the passwords somewhere but it is dangerous to rely on this being secure;
- Users can encrypt passwords but keys should be hidden somewhere;
- Users also can transmit passwords through a network but they have to use secure lines and encrypt transmissions;

It is necessary to mention that there are several types of attacks on software systems with password protection including brute force attacks, dictionary attacks, and some social engineering attacks [STC05]. Besides, visual passwords, which use images or other visual means such as solving a puzzle on the screen, also can be used instead of text strings for authentication. They do not require memorization of some fixed text strings precisely and they are rather human friendly. In addition, there are several biometric (voiceprint, fingerprint, iris scans, DNA, etc.) and smart card approaches. Strong authentication requires a combination of at least two of the main methods of authentication such as passwords, secure ID, and biometrics.

2.1.4 Operating systems security

OS Kernels

Modern operating systems (OSs) support individual users in performing their job and utilise various security mechanisms. Access operations are complex and application specific and users are not interested in the lower level details of the execution of their applications, therefore, this task is performed by the kernel which is the main part of every OS. The kernel has complete access privileges for all machine resources including memory and I/O devices. In contrast to user-level processes which have a restricted address space and execute in the processor unprivileged mode, the kernel runs in the supervisor mode and usually can access any machine register and set up processes' address spaces to protect processes from one another. An OS server can run either as a user-level process or as a kernel process. Sometimes a user-level server can also run in the supervisor mode in order to get access to an I/O device. There are three main approaches to OS kernel design:

- Monolithic kernels – run a large number of OS services as kernel processes;
- Microkernels – shrink the kernel and run a large number of servers as user-level processes;
• Exokernels – a rather radical approach to OS design where the central line of thought is "separate protection from management".

Systems like UNIX (a kernel of around 1M of code and data) are monolithic while QNS, for example, is a microkernel (with only 10K of code). On the other hand, there are systems like MACH 3.0 (500K of code and data) which are somewhere between the monolithic and the microkernel worlds.

Usually, monolithic kernels are not modular which makes the job of implementing new services or modifying the existing ones much harder. Besides, the lack of modularity makes the complete understanding of the whole operation of the system something difficult to achieve. However, that does not mean that it is impossible to build a modular monolithic OS. Anyway, an implementation error in one module of a monolithic kernel can affect other module's data structures and compromise the correctness of all system. Also, modifying one module implies rebuilding of the whole kernel and rebooting of the system. Examples of monolithic kernels are such very popular OS as Windows NT/2000/XP (a microkernel in theory) and various distributions of Linux (however very extendable through dynamically loadable modules).

The microkernel architecture (MacOS X/Darwin, The Hurd, Neutrino) enforces modularity because each server has its own address space and cannot access the kernel data structures directly. Therefore, microkernels are easier to extend and modify because only a small piece of the system should be rebuilt each time and a server can be replaced without interrupting other services or the system functions and without rebooting. However, even though the microkernel approach is more advanced in theory, the large number of context switches involved reduces performance. Moreover, microkernels have problems with concurrency that complicates them significantly.

The exokernel approach is a rather different approach to OS design. It allows application developers to make decisions and have full control over available resources in order to utilise them more efficiently. Even though exokernels are extremely small, they support access control mechanisms, allow applications to request a specific piece of memory, a specific disk block etc, and ensure that the requested resources are free.

**Access control models**

Various OSs also support access control security models such as Mandatory Access Control (MAC), Discretionary Access Control (DAC), Role-Based Access Control (RBAC), Task-Based Access Control (TBAC), Bell-LaPadula (BLP), and Biba models. In general, a security model is a formal description of a security policy. Other two popular models are the Chinese Wall and the Clark-Wilson Integrity model [Wiki07]. All these models have their pros and cons. For example, the BLP model addresses only data confidentiality while the Biba model delivers
data integrity. The detailed description of access control models and their interrelations are presented in Chapter 4.

**Other mechanisms**

There are many other mechanisms which are utilized in OS security: 1) control of OS integrity because the OS is itself an object of access control, 2) modes of operation (i.e., user or supervisor), 3) controlled invocation which allows a user to execute an operation requiring supervisor mode and then switch back after finishing an operation, 4) processes and threads, 5) system stack which is a specially designated part of memory that can be accessed by pushing data on its top or by popping data from its top, and 6) interrupts. Because an OS has to manage access control to data and resources, various OSs support segments, pages, memory protection mechanisms, and multitasking which is employed to interleave the execution of processes belonging to different users. Intrusion detection and prevention techniques can be utilised to strengthen OS security through the use of OS monitors that process, detect, and prevents malicious invocations of system calls [BGM02]. An example of the secure OS specially designed to provide the highest level of security is the Logical Coprocessing Kernel (LOCK) [Smi01].

### 2.1.5 Network security

**Network security (NS) protocols**

The Transmission Control Protocol/Internet Protocol (TCP/IP) is the main protocol that governs the transport and routing of data over the Internet. Other protocols such as the Hypertext Transport Protocol (HTTP) and Internet Messaging Access Protocol (IMAP) run on the top of TCP/IP. Since TCP/IP and many other protocols are not secure enough, therefore, additional protocols for providing network security including Secure Sockets Layer (SSL), Transport Layer Security (TLS), and IPSec are used. SSL and TLS are transparent to software applications and operate on the transport level while IPSec works on the network level.

**Firewalls, IDSs, and IPSs**

Firewalls are employed as a mechanism of preventing unauthorized access to private networks. They are usually used to reduce ability of external intruders to affect internal networks, protect internal networks from others, block access to some resources, monitor and audit communications. Intrusion detection (IDSs) [Axe98,KLC+04,JWZ03] and prevention systems (IPSs) are the second line of defence and usually utilised in conjunction with firewalls and antiviruses.

IDSs are divided into anomaly detection and signature detection (misuse detection) systems. The anomaly detection systems promise to detect abuses of legitimate privileges that cannot
easily be codified into security policy and to detect attacks that are “novel” to the intrusion detection system. Problems include a tendency to take up data processing resources and the possibility of an attacker to teach the system that his/her illegitimate activities are nothing out of the ordinary. Signature-based detection systems promise to detect known attacks and violations easily codified into security policies in a timely and efficient manner. The open issue is a difficulty in detecting previously unknown intrusions. Some researchers [NJW01] present a hierarchical model to support attack specification and event abstraction in distributed intrusion detection. Their model involves three concepts: system view, signature, and view definition. With these three elements, the model provides a hierarchical framework for maintaining signatures, system views, and event abstractions. As a benefit, the model allows generic signatures that can accommodate unknown variants of known attacks. The problem of normal/abnormal behaviour is also studied in [KS94,MY01,BHS+02,LB99]. For understanding and analysis of incidents and network traffic, some researchers suggest applying graphical approaches [CCD+99]. IDSs can be also used for protection and attack detection in databases [LLT02]. Other problems of IDSs are how to evaluate them [LFG+00], how to place intrusion detection devices in the most effective way [Med98] and monitor them [FP99].

For studying attacks, honeynets (honeypots) [Hon05] which are designed to resemble valid systems and collect information about attackers and their methods, can be employed. However, honeynets also can be attacked and compromised [Kra04].

2.1.6 Security evaluation standards

One of the well known security evaluation standards is the Common Criteria (CC) [CC06]. CC is an international standard for a computer security evaluation which is used to evaluate security measures needed for certain software applications. CC harmonises its predecessor including Canadian Trusted Computer Security Evaluation Criteria (TCSEC), the European Information Technology Security Evaluation Criteria (ITSEC), and the United States Federal Criteria (FC). The CC consists of three parts: Introduction and General Model, Functional Requirements, and Assurance Requirements. Functional requirements described in Part 2 introduce the desired security behaviour expected of a target system (i.e., a Target of Evaluation (TOE)) and define security properties that users can detect by direct interaction with the TOE. In other words, functional requirements provide a schema for evaluating IT systems in terms of security requirements. There are 11 Security Functional Classes: 1) Class FAU: Security Audit, 2) Class FCO: Communication, 3) Class FCS: Cryptographic Support, 4) Class FDP: User Data Protection, 5) Class FIA: Identification and Authentication, 6) Class FMT: Security Management, 7) Class FPR: Privacy, 8) Class FPT: Protection of the Trusted Security Functions, 9) Class FRU: Resource Utilization, 10) Class FTA: TOE Access, 11) Class FTP: Trusted Path/Channels. However, the CC are too generic, i.e., do not provide methodology for
security measurements, and just delivers a common set of requirements for the security functions of IT systems and products. Besides, evaluation is a rather costly process in terms of time and money and the result is not necessarily a more secure better product. Evaluation focuses mainly on assessing the evaluation documentation, but not on the technical correctness of the product itself.

2.1.7 Summary
In this section, we summarise open issues in the field of information security. For example, information security mechanisms are not perfect. Ciphers, public/private key cryptosystems, signatures, hash functions can be cracked by brute force attacks, mathematical attacks, timing attacks, etc. The human error factor is one of the most common security problems with information security mechanisms. Visual cryptography has such problems as pixel expansion and the loss of contrast. Quantum cryptography looks very attractive, however, it is rather difficult to implement. Watermarking techniques can be broken by simple waveform processing, detection-disabling methods, advanced jamming/removal, and ambiguity/deadlock issues. Passwords are also vulnerable because users need to store them somewhere but it is dangerous to rely on this being secure. Users can encrypt them but they need to hide keys. They also can transmit passwords through the network but they have to utilise secure channels and encrypt transmissions. Network security has many open issues as well, hence, different secure protocols, IDS and firewalls are applied to protect current computer systems and their communications against such security attacks as DoS attacks, man in the middle attacks, sniffing, and so on. Finally, the Common Criteria specification is used to evaluate security measures needed for certain software applications. However, it is generic and does not provide methodology for security measurements.

2.2 Security in distributed software systems
In this section, we present several distributed technologies including Common Object Request Broker Architecture (CORBA), Microsoft’s .NET, Java 2 Enterprise Edition (J2EE), Peer-to-Peer (P2P), and Web services (WSs), and analyse their security features.

2.2.1 Object-oriented middleware platforms
In this section, we present several object-oriented middleware platforms and their security features. Their detailed description from the non-security perspective can be found in [Emm00].

CORBA and its security features
CORBA [CORBA07,SGM02] is vendor-independent architecture and infrastructure at the same time that computer applications utilise to work together over networks. Using the standard
protocol IIOP, a CORBA-based program from any vendor can interoperate with a CORBA-based program from another vendor in almost any computer, operating system, and network. The CORBA architecture is illustrated in Figure 2.1.

![Figure 2.1. Object request broker architecture [SGM02]](image)

CORBA has the following advantages: easy to invoke objects in different computer languages and extend for new languages; well specified and different ORBs can communicate; IDL provides an abstraction layer over operations; provides many useful services such as Trading Service and Security Service; open standard developed by international consortium. However, CORBA has disadvantages as well: many ORBs do not provide full functionality of the CORBA specification; software code written for one ORB may need modifications for use with another ORB; performance can be sluggish; and the process of development can be quite awkward.

The CORBA security model is a meta-model that is security technology neutral and represents different security approaches, models, and techniques, as illustrated in Figure 2.2. It attempts to generalise many security models and principles and provide a specification that is complete and language independent at the same time. The CORBA security specification is a step above .NET and J2EE which can be considered as an instance of the CORBA security model. Since the CORBA security specification [OMG02] is extensive (400+ pages), we briefly mention some core security services here while the detailed description and analysis of security in CORBA can be found in [DBD+03]. The CORBA security model is rather “expensive” in implementation, configuration and support. In addition, it does not guarantee full protection against attacks [Sie00], however, it is rather flexible and capable to meet different security needs focusing on four aspects of security: confidentiality, integrity, accountability, and availability. These aspects lead to six key building blocks of the CORBA security model including 1) identification and authentication of principals to allow users to be able to provide proof to verify themselves to get access, 2) authorization and access control, 3) security auditing and tracking user’s actions, 4) maintaining availability, 5) secure communications between users and objects, delegation when objects delegate the execution of the operations to other objects, 5) non-repudiation by a third party certify authority, and 6) administrative tools.
Chapter 2 Related work

The model for controlling of interactions of clients and target objects consists of several levels to facilitate secure object invocation by clients [OMG02,DBD+03]:

- Application-level components include client request and target object providing services. The degree of security awareness on this level can vary.

- Different components that realize security such as the ORB core and its services, the ORB security services, and the policy objects with actual security requirements. Figure 2.2 demonstrates two ORB security services including the Access Control Service (ACS), which verifies if the requested operation is permitted and enforces actions for audit, and the Secure Invocation Service (SIS) which connects the client to the target object on the client side and protects the target object in its interactions with the client on the target side.

- The middleware also has to implement some security services and platform specific implementations.

CORBA utilises the Kerberos-based approach for user authentication and access control is based on credentials. Also CORBA as well as .NET and J2EE lack an automatic mechanism for defining and analysis of the security properties of the participating parties.

Figure 2.2. CORBA security model [DBD+03]
Microsoft’s .NET and its security

Microsoft’s .NET [NET05a,SGM02] has been developed as a result of the improving the Component Object Model (COM). The combined result has created XML Web services which allow applications to communicate with each other through the use of such standards as XML and SOAP.

The overall .NET platform architecture contains four main parts:

- .NET infrastructure and tools – Visual Studio.NET, the .NET Enterprise Servers, the .NET Framework, etc.;
- .NET Foundation Services – information sharing services for the Internet, for file storage, user preference management, etc.;
- .NET User Experience – broader, more adaptive user experience, where information is delivered in a variety of ways on a variety of different devices;
- .NET Devices that enable a new breed of smart Internet devices that can leverage Web services.

The .NET framework, as illustrated in Figure 2.3, consists of three key components: the common language runtime which sits on top of operating system services and is responsible for running any downloaded code, a hierarchy of unified class libraries (a hierarchy of object-oriented classes), and ASP.NET, which is a component-based version of Microsoft’s Active Server Pages.

![Figure 2.3. .NET framework [NET05a]](image)

Besides, the .NET Framework includes many security features [NET05b,NET05c,DBD+03]. For example, role-based security is used for managing user identity. Also, ASP.NET provides web application security requirements through the support of HTTP authentication schemes including support for Kerberos, SSL/TLS client certificates, Microsoft Passport authentication, cookies, etc. It also supports traditional methods of performing access control and provides...
URL authorization based on the current user or role. Evidence-based security is used to check all running code and enforces security which helps to run safely software code and control it by the administrator. It is usually handled by the lowest levels of the standard class libraries. In addition, the .NET Framework supports a cryptography library that contains cryptographic functions for encryption (DES, 3DES, RC2, AES, RSA, DSA), digital signatures, hashing (MD5, SHA-1, SHA-256, SHA-384, SHA-512), and random number generation.

Five main security features of .NET are employed to protect the code, data, and systems [MSD06]:

- Code-based access control gives permissions at the code level to access resources based on the application or protection domain that the code is assigned to and verify the evidences of the code identity;
- Role-based access control gives permissions to a user to access resources based on the user’s role in the system;
- Secure code verification and execution (similar to byte code verification in Java) analyses bytecode and checks if the executing code has rights within its certain domain;
- Secure communication encrypts data and messages locally or remotely in a communication channels;
- Secure code and data protection examines that code has not been modified without authorization (cryptographic approaches).

The .NET Security Structural model, as illustrated in Figure 2.4, consists of three parts: CLR, Hosting Environment, and Security Settings. The Hosting Environment is responsible for executing applications and providing the code (via assembly) and its identity (via evidence) in its interactions with CLR. CLR is the core component which delivers a secure execution environment through managed code and code access security. CLR contains the Security...
System which implements the security policy at different levels including enterprise, machine, user, and application domain. Policy files are defined by a security administrator and are based on information provided by the evidence. The Security Settings box contains security requirements at one or more policy levels. The assembly is utilized to identify the required permission set. It is necessary to mention that the .NET security model is rather complex to present it here, hence, the detailed description can be found in [NET05b, NET05c, DBD+03 MSD06].

**J2EE and its security**

Java [Java06,SGM02] is a programming language with many features and capabilities that allow Java applications to run within various operating environments with the Java Virtual Machine (JVM) support. Using Java, developers can create standalone applications and applets that are similar to applications but run within a Java-compatible browser.

The Java platform Standard Edition 5, as illustrated in Figure 2.5, has the following features:

- Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, etc;
- Applets;
- Networking such as URLs, TCP, UDP sockets, and IP addresses;
- Internationalization helps to develop software applications that can be localized for users worldwide;
- Security includes low level and high level algorithms such as digital signatures, public and private key management, access control, and digital certificates;
- Software components (JavaBeans) plug into existing component architectures;

![Figure 2.5. Java 2 Platform Standard Edition 5.0](Java06)
• Object serialization allows lightweight persistence and communication via RMI and Java Database Connectivity (JDBC) that provides access to various databases;
• APIs for 2D and 3D graphics, accessibility, servers, collaboration, telephony, speech, animation, and more.

Figure 2.6. J2EE Platform APIs 1.4 [Java06]

Jini [Jini04] is another Java technology which provides simple mechanisms that enable devices to plug together to form a community without any planning and installation. Jini not only defines a set of protocols for discovery, join, and lookup, but also a leasing and transaction mechanism to provide resilience in a dynamic networked environment. JXTA [JXTA04] employs the P2P paradigm to address a problem similar to Jini’s, i.e., the federation of systems over loosely coupled distributed systems. Jiro is another Java-based technology that focuses more specifically on the domain of managing storage systems. However, Jini and JXTA do not have built-in mechanisms for dynamic adaptation and specification of the security properties of the participating components.

Figure 2.7. The Java 2 Platform - Compile and Execute [DBD+03]
Chapter 2 Related work

The J2EE [JS03,TBL02,DBD+03,Java06] platform, illustrated in Figure 2.6, allows keeping code, data, and systems safe from inadvertent or malicious errors. After the creation of Java bytecode the execution process of Java bytecode involves the class loader (with bytecode verifier), the Java class libraries (APIs), and JVM which interacts with the operating system. They provide a secure runtime environment. “JVM manages memory by dynamically allocating different areas for use by different programs, isolating executing code, and performing runtime checks (e.g., array bounds), which each has a role in insuring security. The block labelled Runtime System as shown in Figure 2.7, contains the Security Manager, Access Controller, and other features that all interact to maintain security of executing code. The Security Manager and Access Controller examine and implement the security policy” [DBD+03]. The platform supports several security specifications including general security (security architecture, cryptography architecture, policy permissions, default policy implementation and policy file syntax, API for privileged blocks, X.509 certificates and certificate revocation lists, security managers and the Java 2 SDK), certification path, Java authentication and authorization service (JAAS), the Java Generic Security Services Application Program Interface (Java GSS-API), the Java Cryptography Extension (JCE), the Java Secure Socket Extension (JSSE). Java and J2EE security is a partial realization of CORBA capabilities.

2.2.2 Security in peer-to-peer systems

Peer-to-peer (P2P) is a technology assigned to network architectures where all nodes have equal capabilities. P2P has advantages over its client/server paradigm counterpart, in that they offer an inexpensive way to exchange information, naturally facilitate distributed computing, provide resilience to security threats, stream centralized server traffic, avoiding performance bottlenecks, and achieve high scalability. A rather full survey of P2P computing technologies can be found in [AS04] where P2P systems are defined: “Peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority”. However, P2P systems suffer from inherit vulnerability rooted from lacking of centralized monitoring and dynamics of peer membership.

P2P systems can be subdivided into as structured P2P (Distributed Hash Table (DHT) (CAN [RFH+01]), unstructured P2P (centralized P2P (Napster [Nap06]), pure P2P (Gnutella [Gnu05]) and hybrid P2P (KaZaA [Kaz06], Morpheus [Mor06], [YG03a])). DHT uses hashing to index peers and files (keys) into the same domain space. All peers and keys are hashed to an identifier space. Then, the special lookup protocol maps a required key
identifier to the IP address of the peer responsible for that key. A storage protocol layered on top of the lookup protocol takes care of storing, replicating, caching, and authenticating the data. In centralized P2P, whenever a peer has a query, it sends the query to the server first, and then after the server replies with the peer ID that stores wanted data, this peer can communicate with the destination. The implementation of such systems is very easy, however, the server is a possible single point of failure. In pure P2P, all participants have uniform roles. Each peer selects a number of peers to which it wishes to connect directly. Pure P2P help to avoid a single point of failure. However, the queries may overload the network and consume its bandwidth. Hybrid P2P (also called super P2P) integrates centralized P2P and pure P2P through the use of super peers that are nodes acting as centralised servers to a subset of clients (peers). Clients submit queries to their super-peers and receive results from them. Super-peers are also connected to each other directly (i.e., similar to peers in pure P2P networks).

Examples of security attacks against P2P systems include man-in-the-middle attack, self replication, pseudospoofing, and shilling [DVP+02]. However, we conclude here that security of P2P systems [DVP+02,AS04] should be strengthened to thwart potential attacks, in particular:

- Anonymity should ensure protection of user privacy;
- Access control through authentication or micropayment systems should protect resource providers;
- File authenticity with reputation-based systems should restrain resource consumers from harmful or wasteful file access.

In addition, P2P security technologies are not mature yet. Research works have been done in the fields of anonymity, access control, file authenticity, and resilience of the network system. Tradeoffs in P2P security seem inevitable, such as tradeoffs between efficiency and anonymity, or between anonymity and file authenticity. Thus, the deployment of P2P network security mechanisms should compromise the practical requirements of degree of security with the price paid for it. The comprehensive list of attacks against P2P systems and countermeasures to resist them is introduced in Chapter 5.

2.2.3 Web services and their security standards

A Web service (WS), as shown in Figure 2.8, is “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards” [BHM+04].
Figure 2.8. The general process of engaging Web services [BHM+04]

Web services have been widely deployed by many companies because of their simplicity of use, platform independence, XML and SOAP support, rich functionality and interoperability. However, Web services also have raised new unexplored security issues and new possibilities of exploiting old security threats. Now, attackers do not need to scan the Web using various network scanners to locate targets. They just check UDDI Business Registry (UBR) to find all the required information for fulfilling security attacks. There are several UBR servers supported by Microsoft, IBM, SAP, and few others that can be employed as a tool for information gathering the attack. The Web service attack tree may include several stages during which the attacker discovers weakness, penetrates Web services, and finally gets access to mission critical applications and infrastructures. As it can be seen, Web services is a rather hot research topic now, however, this technology is rather raw, new, and still evolving. In addition, there are several gaps in SOA [Sle04] that utilises Web services such as reliability, security, orchestration, legacy support, and semantics. Hence, software developers should think carefully when they decide to apply Web services in their distributed software systems. To withstand attacks against Web services, several approaches have been proposed as described below.

[ADD+04] is the strategy paper that defines a web service specification profile WS-I+. WS-I+ builds upon WS-I Basic & Security profiles with the additional specifications: WS-Addressing, WS-ReliableMessaging and the Business Profile Execution Language (BPEL). It also describes and shows how Grid community can choose WS specifications and use them in grid applications now without redeveloping grid applications in future. The strategy to choose proper WS specifications includes five steps: 1) Support from the main industrial vendors, 2) Stability (specifications should be standardized or be nearing the end of the process of standardization), 3) There should be multiple implementations of the specifications from different vendors and the open source community, 4) Available tools to reduce the cost of using the specification, 5) The use of the specifications should not conflict with the use of existing WS specifications.
Web services have security flaws and there are only few security standards which can help to remove them. Several key software vendors including IBM, Microsoft, Sun, Oracle, etc. cooperate with each other to bring new WS security standards. One of the first papers in this area is a joint security whitepaper from IBM Corporation and Microsoft Corporation [IM02] where WS-Security (WSS), WS-Policy, WS-Trust, WS-Privacy, WS-SecureConversation, WS-Federation and WS-Authorization standards and scenarios are described, as illustrated in Figure 2.9. These specifications are built upon foundational technologies such as SOAP, WSDL, XML Digital Signatures, XML Encryption, and SSL/TLS.

Figure 2.9. Web services security specifications [IM02]

WS-Security (WSS) specifies how to attach signature and encryption headers to SOAP messages and demonstrates how to attach security tokens to messages [Web02, OAS04a,OAS04b]. In addition, the WSS specification is a rather low-level standard that does not allow software architects to concentrate on the high-level architecture details and design the whole systems without thinking about low-level components. However, only few of WS specifications have been implemented. Furthermore, each of them mentions very little about the security composition and reasoning regarding the security properties of the participating parties. They do not address interoperation and composition of security services and require a considerable human interference. More WS specifications and the detailed description attacks against Web services and countermeasures are introduced in Chapter 5.

2.2.4 Summary
In this section, we highlight issues related to distributed software systems previously introduced.

CORBA, .NET and J2EE in terms of security
The CORBA security model is a meta-model that represents different security approaches, models, and techniques while Microsoft’s .NET and J2EE provide implementation, at some level, of security concepts and have application programmer interfaces (APIs) with support of security functionality. Actually, the CORBA security specification [OMG02] “emphasizes that
there is no single security model and associated implementation that satisfies all of the
capabilities and features of the CORBA security meta-model” [DBD+03]. .NET and J2EE can
be considered a realization of the CORBA security model. .NET is based on and implemented
only for the Windows OS (we do not take into account an immature implementation for Linux)
while J2EE can be launched on any host with the Java Virtual Machine installed. Both .NET
and J2EE employ the concept of a “sand-box” for protected code execution which is called the
application domain in .NET. CORBA, .NET and J2EE do not have an automatic mechanism for
specifying and analysing the security properties of the participating parties. However, they do
not allow to specify the expected security behaviour and results expected from the composition.

P2P and Web services

P2P security technologies are not mature yet because of vulnerabilities rooted from lacking of
centralized monitoring and the dynamic nature of peers. Besides, peers and P2P routing systems
face external attacks including the man-in-the-middle attack, self replication attack,
pseudospoofing and shilling attacks. P2P systems should be made better in terms of security.
More specifically: anonymity should ensure protection of privacy of users; access control
should protect resource providers through authentication or micropayment systems; file
authenticity with reputation-based systems should restrain users from getting harmful or
wasteful file access. Deployment of P2P security mechanisms should compromise practical
security requirements and the price paid for them, e.g., tradeoffs between efficiency and
anonymity, or between anonymity and file authenticity.

Web services are defined as software systems designed to support interoperable machine-to-
machine interaction over networks. Their interfaces are described in WSDL. Other systems
interact with Web services using SOAP messages in conjunction with other Web standards.
However, few existed WS security standards cannot protect them against security attacks. Many
key software vendors cooperate with each other to implement new WS security standards.
Nevertheless, WS specifications are rather low-level standards which do not allow software
engineers to concentrate on the high-level architectures and design the whole systems without
taking into consideration low-level details.

2.3 The art of creating secure architectures

In this section, we introduce and analyse various approaches for creating secure software
applications and present the area of “Software Security Engineering” (SSE) as well. Actually,
SSE lies in the intersection between software engineering and information security which are
two huge areas researched little. The first book and academic classes on Software Security
appeared in 2001 [McG04]. In addition, researchers have been motivated by works about issues
of software engineering and security [DWW99,Dos01,DS00]. It seems that SSE is still more an Art than a Science or Engineering endeavour.

2.3.1 Introduction

Software systems become more complicated and critical for every domain of the human society [MG06] such as health care, finance services, telecommunications, entertainment, military, etc. These software systems are utilised by various users such as individual users or large companies. However, because of the criticality of information stored by these systems the new issues have arisen related to information security. Traditionally, information system’s security is added to the system after its functional requirements have been addressed. Also, system security features are not systematically designed into the system, and consequently the system may have inherent design flaws vulnerabilities. Hence, companies may spend a lot of time and money to fix everything up. Thus, there is a strong need for a systematic engineering approach to creating secure software systems, especially distributed systems, by considering functional and security requirements at the same time. We call this approach “Software Security Engineering” (SSE). We argue that SSE should use software security knowledge at various stages throughout the entire software development life cycle (SDLC) [BM05], as illustrated in Figure 2.10.

![Figure 2.10. SDLC [BM05]](image)

One of such approaches is Tropos and its modification Secure Tropos [MG07] that allows software developers to take into account security throughout the whole development process of the software system, multi-agent systems in particular. The key point of this methodology is the use of the same concepts on every stage of the development process, and consequently allowing them to validate the final solution employing automated tools and techniques (e.g., a security pattern language, security attack scenarios, formal Tropos language based on Datalog, and an automated tool called T-Tool).

In addition, some researchers try to apply the eXtreme Programming (XP) approach to implement secure software architectures because if XP is adopted for security engineering it
will give higher success rate, better customer satisfaction, short feedback loop, and some other benefits [Bez03]. However, there are some difficulties: there is no roadmap; some XP techniques are controversial (pair programming); there are problems in analysis, testing, and refactoring.

2.3.2 Security requirements

Security requirements should be taken into consideration at the same time as functional requirements [DS00]. It is important to extract accurate trust assumptions during the phase of the requirements analysis [Dos01]. Security requirements have to cover functional security (cryptographic techniques, steganography, etc.) and emergent security which can be implemented using abuse cases [McG04] which specify the system’s behaviour under attack. For creating them, explicit coverage of protection of what, how, for how long, and from whom, is required. Also, for describing security requirements and policies, graphical approaches such as the Language for Security Constraints on Objects (LaSCO) [HPL98] can be rather useful as well.

As previously mentioned, software components can be specified for their security requirements and capabilities, and consequently, they can be combined consistently in such a way to resolve potential conflicts and achieve a wide system’s security objectives. In addition, a component-based software system (CBSS) may include various third-party software components provided in binary form. As the result, software integrators build systems with a rather high potential of high vulnerabilities because they do not have enough information regarding the components’ security characteristics. However, security contracts can define how software components disclose their security properties to each other. Such contracts explicitly express security properties of component interfaces such as ensured and required security properties using logic (e.g., logic programming) and allowing the software integrators assess their suitability and interoperability regarding security in the system. The security characterisation language (SCL) was introduced to solve this issue [KH02,KH03,Kha05]. Since it employs the Common Criteria for Information Technology Security Evaluation (CC) [CC06], BAN logic [BAN90,Wes01], and extended logic programming [DBM01,DEG+01,Rob92], SCL is executable, and consequently allows compatibility checking of security properties between interacting components. This approach [KH02,KH03,Han98] to security-oriented services is different from traditional approaches to system security because security is viewed from a software engineering perspective (i.e., a proactive and predictive philosophy for system security). Furthermore, the security characterisation framework [KH02,KH03] addresses how to characterise the security properties of components, how to analyse at runtime the internal security properties of a system comprising several atomic components, how to characterise the entire system’s security properties, and how to make these characterised properties available at
runtime. The negotiation protocol [HKK04] allows services to establish security-oriented service contracts through negotiation. It can be done automatically and a lot of methods and tools can be involved. A third party can implement its own security methods to protect its components. The approach is summarised in a PhD thesis of K. Khan [Kha05].

The approach seems promising, however, there are some issues. Firstly, a current expression mechanism is not efficient enough, i.e., security properties do not capture all entities involved in security design and analysis. Finally, the composition mechanism is quite simple mapping function calls between a caller and a callee. This mechanism should be improved to allow more than two entities to negotiate their security contracts. Such approach will work even better if security patterns and time properties are employed.

Another open issue is how to ensure that the components as a whole behave consistently and guarantee certain properties. It is not enough to make the composition of the components and force them to work together. Currently, there are few works about prediction of security properties in architectural compositions. However, some researchers introduce a methodology for modelling security system architectures and for analysing them [DWT+03]. The concept of security constraint patterns is presented and it is shown that the methodology is scalable and flexible. This approach is described through a case study, the architecture of the Resource Access Decision (RAD) Facility, an OMG standard for application-level authorization service [DWT+03,YHG+03,YHG+04] and the model hierarchy of RBAC in CORBA access control [GDY+04]. The methodology provides a framework to guide correct implementations of the design from the beginning. The formal method of integrating security administration into software architecture design is also introduced in [YHD+04]. Other approaches are presented in the following chapters.

2.3.3 Analysis of software architectures and security protocols

A formal analysis of software architectures is a very important part of any software system development process [BS99,KH03,FGJ+02]. Hence, software engineers should do it before actual implementing of the system takes place. Also software developers should choose security protocols very carefully. Many modern security protocols have vulnerabilities therefore developers should formally analyse them [BAN90]. These analyses can be done by software and security engineers or by trusted agencies. At the design and architecture level, security architecture should take into account security principles. Also, assumptions and possible attacks should be defined [McG04]. Moreover, risk analysis is a necessary part, hence, risk should be uncovered and ranked by security analysis [DM04]. At the code level, security analysis focuses mainly on implementation flaws. Then, after knowledge is gained it is returned to the development organisation for studying and using in future to create attack patterns [McG04]. Besides, attack languages can be used to develop attack patterns and encode descriptions of
security attacks in a suitable format (e.g., attack signature), then identify these attack
descriptions, and react or report about them [VEK00, EVK02, IDMEF04, Eck01, NASL07,
MM01, Snort07]. Attack languages can be very useful for analysing relationships among
various basic attacks in order to detect more complex security attacks.

2.3.4 Design techniques
Actually, software security engineering approaches can be divided into two types: the injection
and the weaving approaches [Win04]. In the injection approach, software developers think
about functionality first where security is blended in source code and it is rather difficult to add
new security features. Such approach creates a lot of problems and increases cost of applications
in future when security needs to be added. The example of the injection approach is the use of
IDSs [Axe98,KLC+04,JWZ03], firewalls or wrappers [FL96,FBF99] to protect software
applications. In the weaving approach [VBC01,Win04,Fur04,YEY04] security and functionality
are separated from the beginning. Currently, the major of computer systems employ the
injection approach but we believe that the weaving approach is more promising.

Some software architectures [MQR+97] secure the system design in which the different
representations of the architecture of a software system are specified formally. The security
properties of the system are proven to hold at the architectural level. The main ideas are
illustrated by of the X/Open Distributed Transaction Processing reference architecture (X/Open
DTP), which is formalized and extended for secure access control as defined by the Bell-
LaPadula model. This extension helps software vendors to create individual components
independently and with minimal concern about security.

Besides, security patterns [SP07] are applied for designing and implementing secure software
systems so that details of security can be implemented later. One of the first papers that pushed
researchers into this area is [YB97] which contains a collection of patterns (Secure Access
Layer, Single Access Point, Check Point, Roles, Session, Limited View, Full View With Errors)
to be used when dealing with application security. Security patterns have several advantages
[SR01]: 1) Novices can do jobs of security experts; 2) Security experts can understand, point
and discuss security problems and their solutions more efficiently; 3) Problems can be solved in
a more systematic way; 4) Component dependencies can be identified and considered in a
proper way. However, in some cases it is easier to use antipatterns [Kis02] because of
difficulties with proofs that a certain approach is secure.

The Aspect-Oriented Programming (AOP) approach is also utilised to design secure software
architectures [VBC01,Win04,Fur04] through separating security and functionality. It focuses on
crosscutting security and functionality and encapsulating such crosscutting concerns in modules
referred to as aspects. The composition process called weaving allows to integrate aspects with
other modules. The first fundamental AOP paper [KIL+97] introduces aspects and describes the
basics of this approach. It presents an analysis of why certain design decisions (aspects) are so
difficult to clearly capture in actual code, and shows that the reason is that they cross-cut the
system’s basic functionality [Nus04]. Key software vendors (IBM [Sab04] and BEA systems
[Bon04]) and scientists have started researching in this area because in some cases the AOP
approach allows solving some problems better than the Object-Oriented Programming (OOP) or
procedural approaches. In addition, the AOP approach can be applied to mediation security
[YEY04] to decrease design complexity and increase flexibility of the security systems because
mediators allow a mapping of complex models. However, one of the issues is how to protect
data between sources and a system and how to control access. Datalog is used to specify the
mediator functional modules and first-order predicates to specify the security aspects
independently.

Another research group [FGJ+02] proposes an architectural style called the Dual Protection
Style (DPS) for constructing secure software architectures. The style is a collection of rules that
constrain the topology and the behaviour of the components of an application that needs to
support access control, which is based on two principles: user (users can perform only allowed
actions) and code access control (applications can invoke only allowed methods). The formal
specification of DPS is defined in Alloy [Jac02], a formal notation for micromodels of software.
The main advantage of the approach is that it forces software developers to take security into
consideration in the design of the architecture, and consequently reducing the interaction
between software and security engineers. However, DPS has some limitations in distributed
environments if it cannot verify the identity of the requesting service. Moreover, it suffers from
the lack of defence in depth, i.e., if one of defensive strategies fails the other should still prevent
an intrusion.

Finally, it is necessary to mention that IDSs and firewalls can be used to protect software
applications against different security threats. Also security components can be utilized to
protect distributed systems. For such purposes, Commercial Off The Shelf components (COTS)
with CORBA technologies [BS99] can be employed. Wonderwalls also can be applied
(wonderwalls control what objects are accessed on a server and what methods are invoked on
these objects) to protect enclaves. However, such approach is rather slow because “a request
from the client proceeding from Internet goes through the firewall then Wonderwall filters it”.
Additionally, SafeBots [FL96] or wrappers [FBF99] can be used as glue to wrap and protect
components.

2.3.5 Tools and languages

For better understanding, graphical approaches such as LaSCO [HPL98] to visualize data
should be applied. Some authors [KH02,KH03,CH05] use UML, BAN logic [BAN90,Wes01]
and extended logic programming [Das92,Bar03] as a tool to represent security properties of
atomic components and reason about their compositional matching with other components. Others [DWT+03, YHG+03, YHG+04] utilize first order temporal logic to formally represent the system-wide security constraints of the system, Predicate/Transition nets (PrT-nets) to report components and connectors and Datalog. SADL (the Structural Architecture Description Language) [MQR+97], Alloy [Jac02], CC (Common Criteria) [CC06], and SCL (Security Characterisation Language) [Kha05] are also applied.

2.3.6 Security testing

Vulnerabilities are divided into two groups: bugs at the implementation level and flaws at the design level [PM04]. Attackers usually use bugs which are easier to exploit. There are also taxonomies of vulnerabilities. Hence, after the system has been implemented it is important to test its security. Testing for security is different from testing for functionality [Dos01] because security is not an externally observable property. The simplest approach for security testing is to hire a security expert or hacker who tries to attack the system and find vulnerabilities. However, even if the approach is very attractive and effective, the price in terms of time and money can be rather high [STC05]. Hence, it is necessary to develop automated security testing mechanisms. Also testing organizations with their traditional approaches can perform functional security testing. Security testing must include two approaches: testing security mechanisms to check functionality and performing vulnerability and penetration testing to understand and simulate attacker’s techniques [PM04]. It can be done by using white-box (analysing and checking source code and design) or black-box testing (analysing a running program by probing it with various inputs). However, sometimes white-box testing produce reports of potential vulnerabilities where none actually exist.

Penetration testing (a type of black-box testing) using various tools such as Backtrack 2 [Backtrack2], SAINT [SAINT08] or CORE IMPACT [CORE08] is also useful because it gives a good understanding of fielded software in its real environment. However, if it does not consider the software architecture, it probably cannot uncover anything deeply interesting about software risk [McG04].

The problems of testing are also introduced in [DWW99] where open security testing is promoted. [Tho03] describes why security testing is hard. Several types of bugs may escape testing. There is a need for new techniques and tools for design, implementation, analysis and testing. [PM04] argues that testers have to use risk-based approaches and keep in mind system’s architectural reality and the attacker’s logic. By doing this, software security testers can focus on areas of code where attacks can succeed.
2.3.7 Summary

In this section, we highlight several open issues in the field of SSE and introduce directions for future research. As previously said, SSE is a new research area studied little which lies in the intersection of information security and software engineering. Security requirements should be taken into consideration at the same time as functional requirements. Research should be focused on the area of extracting accurate trust assumptions during the phase of the requirements analysis. Software and security engineers together with trusted agencies should do formal analyses of software architectures and security protocols before actual implementing of the system takes place. There are two design approaches including the injection approach, i.e., security is blended in source code, and the weaving approach where security and functionality are separated from the beginning. The weaving approach seems to be more promising, therefore, research should be concentrated on this particular area. Security patterns, anti-patterns, and the AOP approach may help, however, in the AOP approach it is difficult to clearly capture design decisions (aspects) in actual code because they cross-cut the system’s functionality. Four key issues of the AOP approach should be addressed including the identification of aspects, the notations applied to specify aspects, the rules to compose aspects together, and the analysis method of the design product. It is known that humans understand visual information better than numbers and words, hence, graphical approaches to visualize data should be widely employed. Different logic expressions, security policies, and security requirements should be shown as directed graphs. For example, UML may be extended with additional security features. Finally, after the system has been implemented it is important to test its security. The simplest security testing approach called the penetration testing, which is rather attractive and effective, is to hire a security expert or ethical hacker to check system vulnerabilities. However, the price for the approach in terms of time and money can be rather high. Hence, it seems logical to develop automated security testing mechanisms.

2.4 Summary

Software systems are becoming highly complex involving independent collaborating components working together towards achieving the systems’ goals. Besides, security features are integrated into software systems after all functional requirements have been addressed. As the result, these systems have many security flaws and vulnerabilities that can be exploited by attackers, and it costs a lot of time and money for companies to patch them up. At the same time, the new generation of more sophisticated security attacks against these systems is also growing. Therefore, there is a strong need for a systematic software security engineering (SSE) approach for developing secure and robust software systems by considering security and functional requirements at the same time.
This chapter have overviewed and analysed information security techniques and security features of current distributed systems such as CORBA, .NET, J2EE, peer-to-peer (P2P), Web services, and SSE approaches. However, these works mostly use traditional security approaches, i.e., for protecting software systems encryption algorithms that are hard to crack are used or IDSs are employed for detecting security attacks and so on. Also, few researches specify security properties of components in machine readable form and try to analyse them. Besides, structural reconfiguration and adaptation of the software systems are not used to resist various flooding Denial of Service attacks. Other problems include the lack of security ontologies, security patterns, and some others that can be used by current systems to share information about security attacks and countermeasures. Current security models are not really fit to our approach, hence, we develop new one introduced in the following chapters. In addition, this chapter has highlighted several important issues. Finally, research works introduced here cover only several topics related to our approach while the rest is presented in the following chapters.
Chapter 3

SECROBAT: a reference architecture

Software systems have become highly complex, distributed, and mobile involving many various components working together to achieve the systems’ goals. Besides, a system’s security features are usually added after all functional requirements have been addressed, and consequently the system has inherent security flaws. Such software systems may have a large number of vulnerabilities that may be exploited by attackers, and companies spend much time and money to fix them. At the same time, new security attacks, which target these systems, appear. Such attacks as multi-phased distributed attacks or various flooding distributed denial of service attacks are difficult to detect and mitigate. We argue that through collaboration of system’s constituent components, which can organise and adapt their composition dynamically, it is possible to identify and resist these attacks. To communicate to each other effectively, these components need a common vocabulary to exchange security related information (security attacks and countermeasures) among each other.

In this chapter, we present a reference architecture called SECROBAT with hybrid peer-to-peer (P2P) structure and defensive components. This reference architecture is inspired by biological systems and used for developing component-based (or service-based) software systems. Moreover, we define the need for this reference architecture, its key advantages, and how it can be implemented. We introduce how SECROBAT and other parts of our research work including the security ontologies and the GIZKA language are integrated into our model and how they are used for managing security (e.g. security attacks and defenses).

3.1 Biological systems as an inspiration

In this section, we introduce different biological approaches and systems that serve as an inspiration to our approach. Particularly, we describe bacteria and viruses and their mechanisms of using host cells and illustrate that computer pathogens [McA00] behave in a similar way. Also, we present autonomic computing and argue that principles of biological systems can help to solve many problems in computer systems (e.g., issues in software security engineering). We argue that the study of biological systems such as bacteria or viruses can help us to predict and mitigate some security attacks against computer systems and create more effective countermeasures because attackers or malicious software writers may adopt the ideas borrowed from Mother Nature as well.
3.1.1 Toxic and antibiotic resistant bacteria

The microbial world that is invisible by the naked eye consists of fungi, yeasts, phytoplankton, bacteria and other microfauna which grow, reproduce, survive and create ecosystems [De04]. Microbial communities break down organic matters into simpler elements and finally to inorganic molecules. Such process plays a crucial role in the earth’s ecosystems function equilibrium. However, any human or natural activities may create imbalance and affect this equilibrium. Pollution, especially with heavy metals such as mercury (Hg) or cadmium (Cd), is one of the main problems since heavy metals are toxic and have a tendency to be accumulated in soil or water. Toxic resistant marine bacteria [DR07,De04,IKK+02] such as Bacillus and Pseudomonas have internal mechanisms that allow them to adapt and resist degradative activities. Moreover, some bacteria can organise biofilm formations [GMB+03] which may provide additional mechanisms of resistance.

So, if bacteria have various defensive mechanisms, why should not we study them and apply to designing secure software systems? We argue that software and security engineers should utilise the knowledge about bacteria as a source of inspiration for an architectural design. Bacteria already have many analogies in the software world. For example, among analogies for biofilm formations we could name Wrappers [FBF99] and SafeBots [FL96]. Further, bacteria can coordinate their activity as multicellular organisms using intercellular communication (quorum-sensing) [KI00,HJ01]. Such collective behaviour has a lot of advantages such as an ability to migrate to a better environment and adopt new modes of growth. Bacteria use small, self-generated signal molecules called autoinducers to set intercellular communication and regulate the behaviour of bacteria according to population density (analogies are messages or other signal mechanisms).

Also, bacteria differ in the way of their relationships with hosts and other bacteria. Most of the bacteria with quorum-sensing can be amicable (symbiotic bacteria) or adversarial (pathogenic bacteria). For example, such pathogenic bacteria as Pseudomonas auruginosa, which mainly infect individuals who are immunocompromised (HIV), and Burkholderia cepacia, which infect individuals with cystic fibrosis, can cooperate with each other. This symbiotic behaviour provides both species better surviving opportunities. The same symbiosis can be found among computer pathogens such as computer viruses and Trojan horses. Another illustrative example is pathogenic bacteria such as Erwinia carotovora, which causes soft rot in a variety of plants and which has developed a strategy to destroy competing microflora. Some of the computer viruses do the same by trying to find infected computers, and after infecting them with their own copies, trying to destroy other computer pathogens on these machines (i.e., a positive side effect). Also, mutation of bacteria can be considered as an analogy of mutation of computer viruses such as polymorphic computer viruses [STC05].
3.1.2 The spread of computer pathogens on a network

To design and develop effective defensive mechanisms, it is worth to know how such pathogens spread. Biological models can be employed for these purposes. For example, the Code-Red II worm was studied in [MSV+03] for creating a mechanism for mitigating network-borne epidemics. An analysis of a susceptible host population inferred from the Code-Red II epidemic and an empirical Internet topology data set was done in order to simulate how the worm would spread under various idealized defences in the Web. However, the problem of this and other similar models is to validate them because of difficulties with testing and verifying results. For example, researchers from Microsoft have proposed a mechanism of downloading and installing security patches based on the methods used by computer worms for replication and spread [SF07].

3.1.3 Autonomic computing

Autonomic computing, introduced by IBM [GC03] in 2001, combines four major features of biological systems including self-configuration, self-optimizing, self-healing, and self-protection [KC03]. Autonomic computing based approaches can be employed to create attack (especially distributed denial of service and distributed reflective denial of service attacks) resistant and reconfigurable systems. For example, if an attacker or a threat agent performs an attack against a computer system, this system recognizes it and deals with the attack automatically. Computer systems also have to evolve to become self-managing. Self-configuration means that installing, re/configuring a system is not time consuming and error prone and includes the ability for each individual system and the entire IT infrastructure to configure itself on the fly. Self-optimizing allows the system to mutate and improve its own efficiency and performance. Self-healing helps the system automatically detect, diagnose, and repair software and hardware problems. Self-protection allows the system to identify and mitigate attacks and system failures automatically. Reconfiguration and adaptation [CH05] of the system can be used as a defensive mechanism which can be implemented by adding external control modules [CHG+04] organized in a way that the whole system gets self-management capabilities.

To conclude, Autonomic computing is still immature, however, it is aimed to fill the gap between computer systems fully governed by humans on the one hand and biological systems that are fully self-managed on the other hand.

3.1.4 Summary

Our approach involves the features inspired by biology, microbiology in particular:

1. In order to adapt to environment changes and survive, some bacteria employ self-generated signal molecules (autoinducers) to coordinate their activity as multicellular organisms and
create intercellular communication called quorum-sensing. It has inspired us to create the secure message exchange mechanism used by software components to communicate with each other utilising our security protocol, coordinate their reconfigurations in order to survive during security attacks, and deliver more robust component-based software systems;

2. Besides, for creating structural reconfigurations and adaptations of component (service) based software systems, we have adopted the idea of biofilm formations because they demonstrate that bacteria have better chances to resist degradative activities (e.g., heavy metals) and survive in hostile environments when they cooperate and support each other, and change their structure and behaviour as a multicellular organism;

3. The idea of defensive components (DCs) which protect application components and the whole system has been inspired by amicable bacteria. This type of bacteria is able to organise symbiotic formations with other types, i.e., they support each other in order to have better surviving opportunities for both species. In our case, DCs protect application components, which are responsible for providing such functionalities as storage or messaging, from security attacks and computer pathogens;

4. Adversarial bacteria support a strategy to destroy competing microflora. They have served as an inspiration for developing defensive mechanisms, i.e., defensive components destroy computer pathogens and protect the whole system from intrusions. Since developers of computer pathogens borrow many ideas from Mother Nature, we employ them too.

In our approach described in the following sections, we have adopted several key features of biological systems including self-protection, self-configuration, self-optimization, and self-healing.

3.2 Overview of our approach

Software applications become highly distributed and increasingly complex, where various components collaborate with each other in order to achieve system objectives. At the same time, attackers become smarter in creating new types of attacks, especially distributed attacks including multi-phased distributed attacks, which are quite difficult to identify and mitigate. One of the possible solutions to detect and resists such attacks is to allow a system’s constituent components to collaborate and share information regarding various security incidents. To achieve a collaborative defense in distributed environments such as component-based software systems, components should have a common vocabulary to allow them to communicate with each other regarding security attacks and countermeasures. For such purpose, we employ an ontological approach that allows the sharing of a common understanding of information about attacks and defenses among humans and software agents (e.g., defensive components). The process of developing and sharing such a vocabulary is illustrated in Figure 3.1.
As shown in Figure 3.1, a given system (shown as a “cloud”) may consist of various components including application and defensive components (DCs). The application components provide various system functionalities such as data storage or game servers (see Section 3.4). The defensive components operate as a coalition and are used to register security attacks and protect the system. If a DC from this coalition detects a new attack, it adds this attack as a new class to the security attack ontology (SAO) and shares this ontology with other members of the coalition at runtime. When any coalition member develops a defense against this attack, then it adds the countermeasure to the security defence ontology (SDO) as a new class and distributes this ontology among other coalition members. SAO and SDO are supported by other security ontologies including the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), and the security function ontology (SFO), which are presented in the following chapters. Finally, such coalitions can be also very helpful in detecting distributed multi-phased attacks such as Mitnick attacks [UJF+04], mitigating and resisting them.

The model for managing security, as shown in Figure 3.2, consists of several parts including the reference architecture (SECROBAT) (introduced in this chapter), the security ontologies (presented in Chapter 4 and 5), and the formal language (GIZKA) (described in details in Chapter 6). SECROBAT uses GIZKA and security ontologies to specify its architecture while GIZKA is supported by security ontologies. More specifically,
• SECROBAT supports defensive components (DCs) including intrusion detection components (IDCs), honey-pot components (HCs), and key distribution components (KDCs) and the pure peer-to-peer (P2P) [Gnu05] and the super-peer (S-P) [YG03a] network architectures. These network architectures were chosen because they were proved in [AJB00] and [KLS02] to be secure and fault-tolerant, i.e., pure P2P networks are good at surviving attacks while super-peer networks are useful for handling failures. SECROBAT helps to create component-based software systems, which operate as a coalition and are adaptive and reconfigurable in order to identify and mitigate various types of security attacks, and drives the whole research approach. DCs as integrated parts of defensive coalitions can be reconfigured, added or deleted dynamically for protecting the system from intrusions, studying attacks, and securing connections among components. Moreover, coalitions adapt their hybrid P2P/S-P network structures dynamically when there is an attack or the environment changes. For example, during various network flooding Distributed Denial of Service (DDoS) [Mir03] or Distributed Reflective Denial of Service (DRDoS) attacks, a coalition of DCs in addition to blocking malicious hosts also changes its structure from S-P (normal regime) to pure P2P (during attacks) dynamically in order to survive. As mentioned in [AJB00,KLS02], pure P2P networks are better at surviving attacks but super-peer networks are better suited for handling failures. Against other types of attacks such as active and passive sniffing or TCP SYN attacks [STC05], DCs provides such services as registering, resisting, and mitigating security attacks, tracing-back and confusing intruders, protecting components, distributing and updating cryptographic security keys and digital certificates.

SECROBAT can be adopted for designing different kinds of the architectures such as client-server, client-client (peer-peer), or any other hybrid architectures. Hence, on the top of our reference architecture various software applications can be developed including collaborative work or distributed computing systems, Web service-based systems [BHM+04,IM02], P2P systems [Gnu05,YG03a], Grid computing systems [WSF+03], and social networks [Facebook07,MySpace07], or online gaming systems. As previously mentioned, the reference architecture (inspired by biological systems) can be utilised as a reference guide for researching in the field of software security engineering. For example, it can be used to investigate how the components of a system can be organized in such an effective way that components with strong security properties can protect components with weak security properties and as a result the whole system can be more secure (see the example at the end of this chapter).

• Because of the distributed nature of SECROBAT-based systems, their constituent components need to communicate and warn each other about attacks or environment
changes. We apply an ontological approach because we need a way to define our vocabulary that provides us a basis for our approach. Such approach is flexible, scalable, reusable, can evolve over time, solve interoperability problems, and can be used by various distributed components especially DCs. We create several security ontologies briefly mentioned above including SAVO, SASO, SFO, SAO, and SDO. More specifically, SAVO specifies security concepts and binds other security ontologies including SAO, SDO, SASO, and SFO. SASO links high-level security policies and system’s security objectives with low-level technical controls. Besides, it defines security standards, algorithms, concepts, credentials, assurance levels, and security objectives as ontology’s classes which are used by SFO. SAO is utilized as a common vocabulary by a coalition of various DCs which share a common understanding of information about attacks and defenses among each other to ensure better protection. SDO correlates with SAO and is utilised for specifying defensive mechanisms used to identify and mitigate security attacks and describe dependences among security algorithms and standards expressed by SASO and SFO. The detailed description of our security ontologies and their advantages is given in Chapter 4 and Chapter 5.

- To make the process of designing, developing, and managing the distributed systems as simple and flexible as possible, SECROBAT-based systems need to be described in a formal manner at design time as well as managed at runtime. For this purpose, we develop a language called GIZKA supported by our security ontologies and suitable for specifying dynamic software architectures and especially their security properties. This language and its key features are presented in details in Chapter 6.

The reference architecture is going to be discussed in the next sections. The security ontologies and the GIZKA language will be introduced in the following chapters.

### 3.3 The reference architecture

In this section, we introduce the basics of the peer-to-peer (P2P) technology and our reference architecture.

#### 3.3.1 P2P basics

P2P is a technology applied to network architectures where all nodes offer the same services and capabilities and follow the same behaviour. P2P systems is an alternative to the client/server paradigm and has gained much popularity because of its efficient utilization of computing resources such as storage, CPU cycles, content and even human presence. P2P network architectures can be subdivided into structured such as Distributed Hash Table (DHT) [RFH+01] and unstructured such as centralized P2P (Napster) [Nap06], pure P2P (Gnutella) [Gnu05] and hybrid P2P (super-peer) (KaZaA) [Kaz06].
DHT employs hashing to index peer nodes and files (or keys) into the same domain space. In this architecture, all peer nodes and keys are hashed to an identifier space. The lookup protocol maps a desired key identifier to the IP address of the node responsible for that key. A storage protocol layered on top of the lookup protocol takes care of storing, replicating, caching, and authenticating the data.

As depicted below in Figure 3.3, in centralized P2P, whenever a peer has a query, it sends the query to the server first, and then after the server replies with the peer id that stores wanted data, this peer can communicate with the destination. The implementation of such systems is very easy, however, the server is a possible single point of failure.

![Figure 3.3. Centralized P2P](image)

In pure P2P, all participants have uniform roles. Each node chooses a number of peers to which it is directly linked through the network, as illustrated in Figure 3.4.

![Figure 3.4. Pure P2P](image)

This set of connections creates the logical network with a grid topology overlaid on the underlying TCP/IP network. Pure P2P avoids a single point of failure; however, the queries may flood the network and hence induce much workload on it.
Chapter 3 SECROBAT: a reference architecture

Figure 3.5. Hybrid P2P

Hybrid P2P (or super P2P) integrates centralized P2P and pure P2P by introducing super peers, as shown above in Figure 3.5. A super-peer is a node that acts as a centralized server to a subset of clients. Clients submit queries to their super-peer and receive results from it, however, super-peers are also connected to each other as peers in pure P2P style. The relationships between different P2P structures are presented in Figure 3.6.

Figure 3.6. P2P architecture

As previously mentioned, peer-to-peer (P2P) is a generic label assigned to network architectures where all the nodes offer the same services and follow the same behaviour. It appreciably utilizes edge computer resources, in contrast to the conventional client-server paradigm. However, the multi-dimensional security aspect of P2P networks is immature. More precisely, the P2P network architectures and inherit properties of distributed control and dynamics of peer membership may be exploited by attackers who may target peers or routing systems. Inability to preserve requestor’s and provider’s anonymity opens a loophole to compromise user privacy. Therefore, to protect resource providers, access control should perform effective authentication and authorization to stop attackers. Besides, file authenticity should be enforced to protect resource consumers from spread of computer pathogens. Various attacks targeting P2P network infrastructure or semantics demand mechanisms to enforce the resilience property of P2P networks.

The detailed description these security attacks against P2P systems as well as countermeasures to mitigate them are introduced in Chapter 5 and Appendix B.
3.3.2 SECROBAT basics
Currently, many accept the argument that a system’s security is as strong as its weakest component or link, or even weaker. However, in the human society and biological systems, such an observation does not hold. In such a system, for example, the strong can protect the weak, resulting in a system that is stronger than the weakest individual, and even than the strongest individual. Biological systems [KI00] and autonomic computing principles [KC03] serve as an inspiration to our approach, and we argue that software and security engineers can learn from biologists regarding living systems in order to solve many security-related problems in computer systems. For example, corals are able to organise colonies where individual organisms protect each other. Also, such colonies can change their structure if the environment changes or predators attack.

Reference architecture
“A reference architecture provides a proven template solution for an architecture for a particular domain. It also provides a common vocabulary with which to discuss implementations, often with the aim to stress commonality” [Wiki07].

Our reference architecture supports security strategies that help to resist attacks and avoid failures. It consists of application components responsible for functionality (e.g., allow users to play games), defensive components used for protection, and a reconfigurable architecture with a hybrid peer-to-peer (P2P)/super-peer (S-P) structure. It is necessary to mention that the main focus of our research is the defensive components and a hybrid network structure, i.e., we do not discuss much the application components.

Defensive components (DCs)
DCs such as intrusion detection components (IDCs), honeypot components (HCs), and key distribution components (KDCs) allow studying attacks and keeping secure connections among components. They use traditional information security mechanisms but they are exploited in different ways. For example, traditional firewalls/intrusion detection systems (IDFs) are standalone applications used for undertaking particular tasks including detection and blocking attacks. Hence, they may be not effective because they are not integrated into the whole system. Since traditional information security approaches are not fully applicable nowadays for creating secure software systems because they were developed to protect systems which had only few access points to the network while current systems are highly distributed and connected, and allow mobile staff to access systems from any point of the World. Hence in our approach, we follow these modern trends in designing IDCs, HCs, and KDCs which are constituent parts of SECROBAT. Various DCs are described in details in the following sections.
**Structural reconfiguration**

SECROBAT-based systems adapt their P2P/S-P structure dynamically during attacks or when the environment changes, as illustrated in Figure 3.7. For example, during DDoS brute-force attacks traditional approaches use IDSs [Mir03] together with firewalls to detect and protect the system. They are utilised ineffectively because they try to resist attacks by blocking malicious hosts. Because of the distributed nature of the attack, a lot of “innocent” hosts or even segments of the network can be blocked. Such a situation leads to denial of service which is actually the main purpose of the attacker.

In addition to blocking malicious hosts, SECROBAT-based systems also change their structure dynamically from S-P (normal regime) to pure P2P (attacks) in order to allow the system to survive during various kinds DDoS and DRDoS attacks. For instance, an attacker tries to flood a peer (and consequently the whole system) with garbage traffic from multiple computer-zombies. If the system blocks zombies’ IP addresses it may not allow legitimate users to access the network because users and zombie machines may lie behind the same NAT (network address translation) and have the same external IP address. Hence in this case, the system can remove the peer, i.e., remove the main target for attack, and adapt its structure from S-P to pure P2P.

The pure P2P network architecture is characterized as follows:

- There is no centralized coordination;
- Each peer is a client and a server at the same time;
- Peers exchange messages directly (in contrast to the client-server architecture where clients connect to other clients through servers);
- Each peer provides resources and content as well;
- Peers are autonomous (independently join and leave the system).

Also, the S-P network architecture is specified as follows:

- A super-peer is a node that acts as a centralized server to a subset of clients or child peers;
- A super peer keeps an index of all child peers;
- A super peer performs queries on behalf of child peers, i.e., clients submit queries to their super-peer and receive results from it;
- The direct exchange among super-peers is allowed because they are connected to each other as peers in the pure P2P network architecture.

On the top of SECROBAT we can build various software systems including social networks, file sharing networks, collaborative work systems, etc.
3.3.3 Basic design principles

Many current software systems have vulnerabilities due to the lack of proper design and implementation. They are usually developed with only functionality in mind [Dos01, DS00]. Hence, in our approach we take into account both security and functionality during the whole software development cycle.

![Figure 3.7. The structure](image)

We develop several information security principles and apply them to SECROBAT:

- Cryptographic keys are distributed and assigned by KDCs;
- All connections and messages are encrypted by IDCs to prevent sniffing of traffic;
- Messages are time-stamped and signed by components using private keys distributed by KDCs;
- DCs cooperate with each other in order to provide distributed defenses and single points of responsibilities;
- Messages that need the highest level of security are protected with steganography methods [STC05]. For example, very important data can be inserted in a system message body and because of a large amount of such messages there is a rather small chance that an attacker manages to extract any important information even if she sniffs network traffic and then decrypts messages.
- Mirroring and replication of components are implemented to restore systems after attacks.
- As shown in Figure 3.7, components such as defensive (IDCs, HCs, KDCs) and application (e.g., storage servers – SSs or game servers – GSs) components can grouped together to create peers or super peers (P1, P2, P3, SP1 and SP2). Some of peers can be super-peers (SP1 and SP2), which act as centralized servers to a set of clients and as peers to each other [YG03a]. In fact, a single component also can be treated as a peer hosted on a single computer. On the other hand, a single computer can also host multiple peers.
- Every peer in SECROBAT has a plan about what to do during and after attacks (because the main goal of the SECROBAT-based system is to survive and provide required
functionalities), which is specified in our GIZKA language, introduced in Chapter 6. For example, some peers can imitate that they are “dead” and some can try to resist attacks or forward all malicious network traffic to HCs. Peers also know how to recover after attacks because super-peers store states of peers and the whole system itself.

The system gathers and manages information about itself and about the system environment using components that serve as probes. After the evaluation of such information by managers who can be a human or software agent, they decide what should be done in order to improve performance of the system or survival during attacks (e.g., structural reconfiguration of the system).

3.3.4 Defensive components
SECROBAT supports two types of components including application and defensive components. Application components such as storage components are responsible for functionality of the system while defensive components such as HCs, IDCs, and KDCs provide security functionality. Defensive components (DCs) apply traditional security techniques utilised by intrusion detection systems (IDSs), honeypots and key distribution (KD) systems, but they are utilised differently. They are integral parts of the entire system while in traditional systems IDSs are used as stand-alone applications. Thus, such approach allows us to have better control over the whole system.

Honeypot components
HCs or fake components are used to confuse attackers. They pretend to be application components but in reality they shadow them, create honeynets (a trap used for detecting, deflecting, and gathering information about attackers) [PH07,Hon05], and register attacks. HCs also trace-back intruders, behave as spies, and study attacks. They are linked to the system through IDCs and employed during attacks when IDCs forward malicious traffic to them.

Key distribution components
KDCs are responsible for providing and updating cryptographic keys (symmetric private keys and pairs of asymmetric private/public keys) and digital certificates to various system components. Keys are protected by IDCs and utilised to encrypt/decrypt and sign messages.

Intrusion detection components
IDCs are employed to register attacks in source networks, victim networks and intermediate networks; prevent, detect and mitigate attacks; trace-back intruders; and protect components. IDCs are distributed over the network and analyse network traffic and behaviour of components, and work as spies or wrappers to cover and protect “weak” components. Some IDCs operate as
ordinary firewalls and filters by blocking or filtering network traffic from suspicious peers or components. Other IDCs use antivirus techniques for protecting SECROBAT from network pathogens such as viruses, worms, or Trojan horses. Besides, every IDC looks after other DCs because intruders can try to attack protection mechanisms.

After an attack is detected, IDCs can follow several strategies to minimize and deflect the damage from the attack including sending signals to the network to reconfigure it, filtering the network, blocking attack agents, etc. Besides, IDCs log attacks and analyse them to create traffic patterns and trace-back intruders. Also they examine protocols and application components utilised in the system. Since network traffic in SECROBAT-based systems goes through IDCs (IDCs act as proxies), it allows systems to have better control over their various system parts.

3.3.5 Reconfigurable architecture
Well-designed architecture saves a lot of time and money because many application bugs and design flaws can be avoided from the beginning.

Structure
SECROBAT utilises a hybrid structure where software components are organized into peers. It can change its structure dynamically during the normal regime and during attacks (see below). As mentioned above, SECROBAT unites two types of structures: pure P2P and super-peer (S-P). The pure P2P network is homogeneous where most peers have approximately the same number of links while the super-peer network is inhomogeneous with the majority of the peers have one or two links but only few peers have a large number of links, guaranteeing that the system is fully connected. By combining these two types of network topologies, we have the robustness of super-peer networks to system failures coupled with the robustness of pure P2P networks to attacks. It is proved in [AJB00] and [KLS02] that pure P2P networks are better at surviving attacks while a super-peer topology is ideal for handling normal operation including query forwarding and surviving random failures. Hence, reconfiguring from S-P to P2P allows the system to survive during various security attacks including DDoS and DRDoS attacks but at the same time the system mainly looses in performance because pure P2P systems generate too much broadcast network traffic than S-P systems do. On the other hand, when the system changes its structure from pure P2P to S-P it certainly gains in performance but loses in capability to survive and withstand security attacks.

Two regimes
In the normal regime, as illustrated in Figure 3.8, every peer is connected to one of the super-peers. However, if there is an attack (e.g., a flooding DDoS attack) on the system its structure changes, as shown in Figure 3.9.

During this attack, peers try to connect to as many peers and super-peers as possible, instead of connecting to only one of super-peers. For example, the temporal storage server starts behaving as peers and is still responsible for the search within the system but needs to cooperate with other peers (see Figure 3.9). Malicious traffic is blocked or forwarded by IDCs to honeynets for studying.

In the remaining of this section, we will talk about different types of peers and the roles they play in our approach.

**Temporal servers**

A temporal server or so-called chimera (a term from microbiology) is an abstraction used to describe situations when the system needs additional resources to increase performance or survive. Chimeras can accommodate resources of various peers fully or partially. For example, chimeras can represent peers that behave as temporal storage or index servers for a limited period of time to allow the system to improve performance. Index servers can store a list of
neighbouring peers that can be used to improve search speed in the network if current search speed is not adequate (search requests can overwhelm a pure P2P system because they are broadcasted to all neighbouring peers). When the system needs additional resources such as the storage space, it borrows them from other peers (if they allow) and incorporates these resources to chimeras. For example, there is an online gaming system that needs additional CPU resources because of an excess of game players. The gaming provider can just buy additional servers (not a flexible approach since these hardware resources may not be needed in future) or can just borrow some CPU time from game players who want to exchange it for time they are allowed to play the game (more flexible approach). However, temporal servers may be the main target for security attacks because they may store system data or private information about game players.

**Fake peers**

IDCs together with HCs can be organised into peers called fake peers used to confuse intruders similar to honeypots [PH07]. For example, there is the online gaming system with several storage servers utilised to store private data of game players such as names and bank account details. To confuse attackers, the game provider creates several fake peers that behave as not properly configured and protected storage servers while real storage servers are fully secured and hidden. If an attacker breaks a fake peer she may find a lot of fake information, and at the same time, she can be studied, traced-back, and all her actions can be recorded in order to provide evidences for law enforcement or legal authorities to deal with her in future.

**Replicated peers**

Peers with important data can be replicated and repaired when some network segments are blocked or destroyed. Such strategy allows the system to be robust and is supported by reserved peers that “sleep” and do not participate in the network activity. Each peer has a significance number and if this number is zero it means that the peer does not need to be replicated. If this number equals one or two, it means that the peer should be replicated once or twice. Super-peers behave in a similar way. For example, in the case of the online gaming system, the replication mechanism helps the system to survive during security attacks. Peers that are responsible for storing user accounts are replicated and available all the time, however, the cost of the replication (i.e., computer resources) can be high.

**Reserved peers**

Super peers support the index table of reserved peers which are utilised when peers need to be replicated. Also, every super-peer stores states of its peers and copies of the states from other super-peers. These states are used during and after attacks to restore “dead” peers. For example
in the case of the online gaming system, upon detecting the Smurf attack [SP08, Wiki07] against its peers, peers begin replacing their “dead” neighbours with replicated peers. The Smurf attack happens when the attacker sends packets to the system with the source address spoofed to the victim’s IP address. States of “dead” peers are taken from super-peers and then copied to replicated peers. This process of replacements continues till the end of the attack or till the number of replicated peers becomes zero. At the same time, if the difference between the significance number of the attacked peer and the real number of its replicated peer is greater than zero, the system tries to create new copies of the attacked peers and give time to IDCs to block malicious network traffic. Once the system does not detect any attack, it returns to the normal regime and tries to restore “dead” peers. As the part of the normal regime, it rebuilds the index table of reserved and replicated peers to employ them in future.

Security contracts
As previously mentioned, peers need to communicate with each other. However, they have to satisfy security requirements and capabilities of each other governed by security contracts defined in our GIZKA language presented in details in Chapter 6. Without satisfying these security contracts, peers are not allowed to communicate.

When a peer wants to connect to another peer, it sends an initial message that contains security requirements and capabilities. If the peer responder’s security capabilities satisfy the peer requestor’s security requirements and vice versa, then a session between these two peers is established. This session is valid for the certain period of time and during this time the security contract is verified in certain time intervals. If the session expires or is terminated for some reason then a new session should be initiated.

3.4 Example gaming system
In this section, we apply SECROBAT to the example online gaming system to demonstrate our approach.

3.4.1 Introduction
The computer game industry has grown rather fast for the last several years especially in such areas as online games for desktops, game consoles and mobile phones. One of the biggest markets is China’s online gaming market with revenue at US$298 million in 2004, US$467 million in 2005 and US$1.3-2 billion in 2009 [Nys05]. It does not even include the sale of virtual gear for game characters, such as clothing, which is also a fast growing business in China. The mobile gaming market has become the “secret weapon” for many companies in China. Faced with huge potential, industry players try to take advantage of the expected growth. However, due to the complexity of online gaming systems, piracy and other software security
threats such as attacks on game servers and game players, game companies have to develop strategies and technologies to prevent security attacks and protect their clients. They aim to help build trust relations between players and game providers while deriving a healthy profit. Furthermore, since small game companies have limited budgets and cannot compete well with large game providers, they have to develop strategies to implement and maintain secure and robust gaming systems with a limited number of available resources.

3.4.2 Main issues

In this section, we highlight several issues related to the online gaming system’s security that are answered in the following sections.

• Because of computer piracy, it is very difficult to make a profit by selling copies of online games. Hence, one strategy would be to distribute them for free while game players would pay only for the time spent during playing the game. Therefore, online payments have to be secured since game players do not want to lose their money.

Questions: From the system perspective, who should be responsible for protecting online payments? What system components should do it and how they should be organised in the most effective way to perform their tasks?

• If a game becomes quite popular then it attracts a large number of game players which can overwhelm game servers. Also, game servers as well as some other gaming system components can be attacked by using different types of attacks such as sniffing or various network flooding DoS/DDoS/DRDoS attacks which deplete network bandwidth. A traditional approach to resists such attacks includes buying additional hardware resources such as servers and network bandwidth. However, it can be not economically feasible for small game companies as well as for large ones because such hardware resources may not be used in future. Hence, there should be other approaches to overcome this issue. One of them is to allow the gaming system to reconfigure itself dynamically in order to resist against security attacks and distribute load among game components. Some system components (e.g., defensive components) should be specifically designed to protect the online gaming system.

Questions: How can game providers, especially those with the limited funds, solve the issues described above? What kind of software systems should they adopt? How should such systems be organised in order to withstand security attacks? How should system components be designed and operated?
Game providers need a lot of economic investments, especially at the beginning. Small companies with small budgets simply are not able to start the business because, except of paying for game licenses, they have to pay for hardware, additional software applications such as firewalls and intrusion detection systems, technical staff, etc. Hence, there should be a way to run game servers that support as many users as possible while using a small amount of hardware resources. The gaming system should be easily managed by a small number of technical specialists. Moreover, the system should be secure and robust and utilise security standards that are free and used by many companies because the system has to interact with many parties such as game players or finance organisations.

**Questions:** How can game providers satisfy such different requirements? What kind of security techniques should be used in order to allow game providers to easily manage, analyse, and reason about a system’s security?

Furthermore, in order to be attack-protected and robust, the online gaming systems should be designed and implemented using proven security techniques and principles, incorporating appropriately defensive components in system design and adopting adaptive and reconfigurable architectures. In the next section, we present the gaming system with such features designed following the SECROBAT approach.

### 3.4.3 Gaming scenario

If a game player (Client1), as shown in Figure 3.10, wants to play a game called Heroes of Might and Magic (HMM6) for the first time, she downloads a game client software program from the website www.hmm6.com. After installing it, she specifies that she wants to play HMM6 with game players from Oceania in their 20th from 9am till 5pm. A system manager (Manager1) checks if Client1 has enough funds to play and decides what security policies should be enforced for Client1, and what security techniques (IPSec and 3DES) should be used because security measures may reduce performance or network bandwidth. Manager1 stores all its data on a storage server (SS1). In addition to playing games, game players can do other collaborative activities including chatting or conducting videoconferences. Every game player can make a payment that goes through the nearest IDC and is checked and forwarded to the bank.

### 3.4.4 Architecture

The SECROBAT-based gaming system in the normal regime, as illustrated in Figure 3.10, is the super-peer system where the majority of peers have one or two links but a few peers have a large number of links.
All connections between game players (clients) and the system (game servers (GS1, GS2, and GS3) and storage servers (SS1, SS2, and SS3)) are controlled by intrusion detection components (IDC1 and IDC2). A key distribution component (KDC1) is responsible for distributing keys and certificates in order to maintain secure connections among different parts of the system. System components as well as game players link to each other through IDCs that are peers for other IDCs and super-peers for anyone else. A honeynet peer (HC1) is used to collect new data for intrusion detection attack patterns, study attacks, and attract intruders by exposing well-known vulnerabilities. Storage servers (SSs) are responsible for storing personal data of game players, system security properties and policies, etc. Besides, game servers (GSs) provide game services and store game states. They are connected to other GSs directly (i.e., pure P2P network architecture), however, they are linked to game players through IDCs. Reserved peers (Reserved1) have been presented are used for adding additional recourses during attacks, as described later. The main task of Manager1 is to organize other components and peers and manage the system. The system is designed in such a way in order to allow IDCs to protect systems components from external and internal attackers. Besides, KDCs are distinguished from IDCs and HCs because the distribution of cryptography keys can be a rather resource intensive task. Finally, the manager is separated from other components in order to reduce the load on such component and govern the system more effectively.
3.4.5 Analysis of security features

In this section, we explain how the system design described above addresses the security issues highlighted earlier and summarise them.

- The network connection between the system and the external bank is encrypted and inspected by IDC2 in order to protect online payments of the game players. It is difficult for an attacker (Attacker) to break communications among Client1 and other peers because IDC1 and IDC2 detect cases of intrusion, study log files, and use information about the studied attacks from HC1. Messages are also encrypted using cryptographic keys provided by KDC1.

- When Client1 is attacked, IDC1 tries to protect it and forwards malicious traffic to HC1 for studying the attack and tracing-back to Attacker. To withstand flooding brute-force DDoS attacks such as Smurf or Fraggle (described in Chapter 5), Manager1 orders the system to changes its structure, as illustrated in Figure 3.11. Besides, the additional IDC3 is added using the Reserved1 peer in order to resist attacks and protect system components and clients. Also, IDC1, IDC2 and IDC3 look after each other and HC1 in order to detect and mitigate security attacks against defensive mechanisms that are always active. Furthermore, clients connect to other clients directly as in the pure P2P networks (e.g., Client1 and Client2 can still play the game even during attacks). Also, if a game provider needs more resources, the system can try to distribute load dynamically and evenly. To minimize expenses, a game provider uses Manager1 to control the system and utilises Client3 and Client4 as game servers. However, game players on Client3 and Client4 should not have access to the system or other users’ data. Hence, the system data stored on these clients is encrypted.

- The system can use a security description and reasoning scheme that supports security properties expressed in GIZKA (described in Chapter 6). If a peer hosts Web services, these security properties can be expressed in the form of OWL [OWL07], OWL-S [OWLS07], and SWRL [SWRL06]. Moreover, clients’ computer resources such as the CPU time or the storage space can be borrowed in exchange for the game time to create temporal servers (called chimeras) in order to allow the small game providers with the limited funds to compete with the large one.

Now, we summarise the main features of SECROBAT introduced in the previous example:

- Defensive components are used to protect the system against attackers, control online traffic, cooperate with other components, identify and study attacks, and distribute cryptographic keys. They use ontologies as a vocabulary to share information about security attacks.

- The manager is employed to govern the whole system and its constituent components.
In order to mitigate the traffic flooding attacks, the system changes its structure.

In order to get additional resources, the system can use game players’ computers as game servers. Besides, reserved peers are employed for creating new defensive components.

3.5 Summary
Currently, software applications become highly complex and distributed, involving different components that collaborate for achieving certain system goals. However, attackers also become smarter and more professional in developing new types of security attacks, distributed attacks in particular. Many of these attacks including multi-phased distributed attacks can be prevented, detected, and mitigated only through the collaboration of a system’s components. Besides, to achieve such collaborative defense in various distributed environments, especially in component-based or service-based software systems, their constituent components should have a common vocabulary to allow them to share information with each other about security attacks and defenses. Also, these systems should provide an architecture that can be used to support and distribute this vocabulary among other components. Moreover, this architecture should be specified in a formal way in order to be secure, robust, easily managed, and flexible.

Hence in this chapter, we have presented such architecture (reference architecture) called SECROBAT for managing security, security attacks and defenses in particular, which allows SECROBAT-based software systems to be secure and robust. We have described its key features including basic design principles, the reference architecture, defensive components (DCs) including intrusion detection components (IDCs), honeypot components (HCs), key distribution components (KDCs), and the hybrid pure P2P/super-peer (S-P) structure. IDCs are adopted to detect and mitigate security attacks, protect internal system components and other DCs, and serve as proxies between internal system components such as DCs and application components and the external world. HCs are employed to confuse intruders imitating to be application system components that provide various system functionalities such as messaging or storage while in reality they can be organised in fake peers used to shadow real resources and to study security attacks. KDCs are only responsible for generating and distributing cryptographic keys and digital certificates among system’s components.

SECROBAT uses the hybrid pure P2P/super-peer (S-P) structure. The S-P network architecture is employed during normal system operation since it is proven to be robust to system failures while the pure P2P network architecture is used during attacks because of the robustness to security attacks. SECROBAT allows to switch from one network architecture to another one depending on a situation (security attack or normal regime).

Other key features of SECROBAT are temporal servers called chimeras, replicated and reserved peers, and security contracts. A chimera is an abstraction or virtual peer that represents
situations when a CBS needs additional resources. Chimeras can accommodate resources of various peers for a limited period of time and provide these resources to those parts of a system that really need them in order to increase performance or survivability of the system. Replicated and reserved peers are simply added for redundancy and providing backup features to allow the system to survive during security attacks. The process of the message exchange among components is governed by security contracts specified in GIZKA (described in Chapter 6) using security ontologies (presented in Chapter 4 and Chapter 5).

The reference architecture can be implemented using various technologies including P2P, Web services, and Grid computing. Also, SECROBAT can be adopted for developing various software architectures such as client-client or client-server. Moreover, on the top of it different software applications can be built including P2P and Grid computing systems, systems for collaborative work, social networks, or online gaming systems, and so on.

To define a vocabulary, we have adopted an ontological approach that allows sharing a common understanding of information regarding security attacks and countermeasures among DCs, software agents, or humans. Our security ontologies and the GIZKA language that specify SECROBAT in a formal way are described in details in the following chapters. Together with SECROBAT they represent a model for managing security. Finally, it is necessary to mention that biological systems reported above have served as an inspiration to our approach.
Chapter 4

Security ontologies

Software systems have become increasingly distributed, involving many independent and collaborating components working towards achieving system goals. At the same time, the number of new security attacks, which are quite difficult to identify and mitigate, against these systems is also growing. In the previous chapter, we have argued that one way to detect and resist against such attacks is through the collaboration of a system’s constituent components. To achieve collaborative defense in a distributed component-based software system, a common vocabulary is needed for the components to communicate and work with each other in detecting attacks and deploying countermeasures. We adopt an ontological approach to establishing such a common vocabulary and introduce ontologies concerning security attacks and defenses. The security ontologies specify the security concepts and their relationships in a way understandable to both humans and software agents. In this chapter, we present our security ontologies, which link high-level security policies and system’s security goals with low-level countermeasures. Also, they help to separate security requirements (i.e., “What”) from their implementations (i.e., “How”) matched on required actions (i.e., “Do”), and explain the reasons of choosing them over other classifications. We describe in details the security asset-vulnerability ontology, the security algorithm-standard ontology, and the security function ontology. In the next chapter, we present the security attack and defence ontologies utilised as a common vocabulary of security attacks and countermeasures and employ a case study involving Mitnick attacks to demonstrate how the security ontologies are utilised in order to detect and counter attacks.

4.1 Introduction

In recent years the development of ontologies, explicit formal specifications of the terms in the domain and relations among them, has been moving from the realm of Artificial-Intelligence laboratories to the desktops of domain experts [NM07]. For example, the WWW Consortium (W3C) developed the Resource Description Framework (RDF) [BG99], a language for encoding knowledge on Web pages to make them understandable to electronic agents searching for information. The Defence Advanced Research Projects Agency (DARPA) in conjunction with the W3C created DARPA Agent Markup Language (DAML) and its successor Web Ontology Language (OWL) through extending RDF with more expressive constructs aimed to facilitate agent interaction on the Web [HM00]. Thus, many disciplines now develop
standardized ontologies that domain experts can use to share and annotate information in their fields. For instance, medicine can explicitly be taken as an example that has produced large standardized structured vocabularies such as SNOMED [PS00]. So, an ontology defines a common vocabulary for domain experts who need to share information in the domain that encompasses machine-interpretable definitions of basic concepts within this domain and relations among them.

While there are several similar definitions of ontologies [NM07], we follow the one given by [UKM+98] who specify the term “ontology” as follows: “An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms.”

As previously mentioned, many research works apply an ontological approach to different knowledge domains due to its advantages over taxonomies and other classification schemes. From the above definition, we identify the following features of ontologies:

- Achieving a shared understanding of structured information, which can be reasoned and analysed automatically, among both humans and software agents;
- Specifying various semantic relationships among different concepts;
- Solving interoperability problems;
- Being reused and evolving over time.

In this work, we develop security ontologies which specify information security issues, including security attacks and defenses in particular. The main security ontology called the security asset-vulnerability ontology (SAVO) illustrates how vulnerabilities are exploited by intruders in order to perform attacks against hosts (or systems) that may affect their assets (anything that should be protected within any environment) safeguarded by defensive components. SAVO links high-level concepts such as security policies and security goals with low-level technical countermeasures in order to separate security requirements (“What”) from their implementations (“How”) matched on required actions (“Do”). Moreover, SAVO binds other security concepts, mechanisms and ontologies including the security attack ontology (SAO), the security defence ontology (SDO), the security algorithm-standard ontology (SASO), and the security function ontology (SFO) for defining information security issues and assisting developers to create better and more efficient protection against system attacks. SASO connects system’s security objectives and high-level security policies with low-level technical solutions and specifies security algorithms, standards, concepts, credentials, assurance levels, and security objectives as ontology’s classes employed by SFO. On the other hand, SAO is utilized as a common vocabulary by a coalition of various defensive components (for example, intrusion
detection components) which interact with each other and share a common understanding of information about attacks and defenses to ensure better protection. SAO closely correlates with SDO, which is mainly used as a specification of defensive mechanisms to resist certain security attacks and define dependences between the security algorithms and standards expressed in SASO and SFO. It is worth mentioning that such separation of ontologies is done for simplicity. Each class of SAO has the property which relates to the certain class of SDO. Using this property it is possible to specify the rules of anti-correlation (i.e., a selection of proper countermeasure to mitigate a security attack). Besides, every class of both ontologies has the property which may be used to describe the particular attacks and defences. In case of SAO, this property may also include the rules of correlation rules which allow to identify security attacks that are possibly related to each other and worm out the triggering root attack. These security ontologies and some of the rules are presented in the following chapters. Furthermore, the security ontologies are employed in our security language introduced in Chapter 6.

4.2 The security asset-vulnerability ontology

Below, we briefly introduce several security terms used in our ontologies and explain their relations through the use of SAVO illustrated in Figure 4.1. Some of the security concepts, are not fully presented here, but are described later.

SAVO is built on top of other security ontologies such as SASO, SFO, SAO, and SDO. SAVO is a high level security ontology that depicts information security in a simplified manner especially for non-security professionals and its design is based upon our expertise and
knowledge of information security and the data extracted from [STC05] and [KLK05, DNT04, UJF+04, MM06, SH05]. SAO, SDO, and SFO are represented by the classes ‘Attack’, ‘Defence’, and ‘SecurityFunction’ respectively. The dependencies between SAVO, SASO, and SFO are shown later. Similar to the NLR security ontology [KLK05], our security ontologies are designed to support several features:

- The ability to specify security related information for different types of resources in various level of detail for different environments.
- The ability to be reusable and easily extendable.
- The ability to allow mapping between high-level and low-level security requirements and capabilities.

We first introduce several security concepts needed for better understanding of provided information. The detailed description can be found in [STC05]. We start from the term “asset” (the class ‘Asset’) which is anything within any environment that should be protected such as data (the class ‘Data’), software (the class ‘Software’), accounts (the class ‘Account’), and resources (the class ‘Resource’), as illustrated in Figure 4.3. The class ‘Asset’ consists of several subclasses including ‘ClientData’ and ‘SystemData’ which refer to client and system data; ‘Component’ and ‘Service’ that are related to the software implementation; ‘CPU’, ‘Memory’ and ‘Storage’ that present resources; ‘ClientAccount’ and ‘SystemAccount’ that are the subclasses of the class ‘Account’. However, to guarantee the certain level of protection for the assets several high-level security objectives (the class ‘SecurityObjective’ from SASO) should be applied as described in the next section.

The term “threat” (the class ‘Threat’ in Figure 4.1) is any occurrence which may cause any unwanted outcome for a company. A threat agent (the class ‘ThreatAgent’) is an agent that can use a threat in order to exploit vulnerability, while vulnerability (the class ‘Vulnerability’) in turn is the absence or the weakness of defence (error, flaw, etc). Further, risk (the class ‘Risk’) is the possibility that a threat agent will exploit vulnerability to damage an asset. Exposure (the class ‘Exposure’) reveals the possibility that ‘Vulnerability’ will be exploited by ‘ThreatAgent’. Besides, ‘Vulnerability’ may also result in ‘Exposure’. Furthermore, every instance of ‘Exposure’ is a risk (the class ‘Risk’) which can be mitigated by using a safeguard (the class ‘Defence’) utilised to resist attacks (the class ‘Attack’). Also, ‘Defence’ applies security techniques and mechanisms (the classes ‘SecurityFunction’ and ‘SecurityAlgorithmStandard’ represent SFO and SASO). The more detailed report on the classes ‘Attack’ and ‘Defence’ that represent SAO and SDO can be found further in the next chapter.

A threat agent may use a threat to perform ‘Attack’ and endanger an asset. A threat agent exploits vulnerability for attacking a host (the class ‘Host’) which hosts assets (a peer is a victim
of an attack). A security attack, which has a precondition (the class ‘Precondition’) related to vulnerability, causes a security event (the class ‘SecurityEvent’).

![Diagram of security classes](image)

**Figure 4.2. The class ‘Consequence’**

A security event can be triggered by a system failure (the class ‘Failure’) as well. Besides, a security event may affect an asset and result in a consequence (the class ‘Consequence’) which affects an asset too. A consequence may be a destruction of data (the class ‘DataAnnihilation’), an information leakage (the class ‘InformationLeakage’), an illegal access (the class ‘IllegalAccess’) or a DoS attack (the class ‘DoSAttack’), as depicted in Figure 4.2.

![Diagram of asset classes](image)

**Figure 4.3. The class ‘Asset’**

An asset (see Figure 4.3) has a value (the class ‘AssetValue’) which can be estimated through exploiting both the quantitative risk analysis (the class ‘QuantitativeRiskAnalysis’) and the qualitative risk analysis (the class ‘QualitativeRiskAnalysis’), as shown in Figure 4.4.

![Diagram of asset value classes](image)

**Figure 4.4. The class ‘AssetValue’**
An asset can contain multiple vulnerabilities which can be explored with other vulnerabilities, as illustrated in Figure 4.5.

![Diagram of Vulnerability Class]

**Figure 4.5. The class ‘Vulnerability’**

The class ‘Vulnerability’ has properties including a vulnerability name (the class ‘VulnName’), an attribute (the class ‘VulnAttribute’) and associated with them values (the class ‘VulnValue’). A patch (the class ‘Patch’) removes vulnerability and is developed by a supplier (the class ‘Supplier’).

As an example of specific techniques that support SAVO, we introduce the quantitative and qualitative risk analysis. The quantitative risk analysis provides methods to estimate a concrete probability percentage (calculator based). There are several functions associated with the quantitative risk analysis [ISACA08]:

- The exposure factor (EF) shows the percentage of loss that a company would experience if a certain asset were violated by a realized risk. This exposure factor is expressed as a percentage (%);
- The single loss expectancy (SLE) is the cost of loss calculated using a formula. SLE is expressed in a money value ($) and is calculated using the formula below:

\[
SLE \[\text{\$}\] = AV \times EF
\]

where AV – asset value;
- The annualised rate of occurrence (ARO) is the expected frequency with which certain risk or threat will occur within one year. ARO is expressed utilizing the formula below:

\[
ARO = Q_o / \text{year}
\]

where \(Q_o\) – quantity of occurrence;
- The annualised loss expectancy (ALE) is the possible cost per year of all instances of a certain realized threat against a certain asset. ALE is counted using the formula below:

\[
ALE = SLE \times ARO = AV \times EF \times ARO
\]

- The annual cost of the safeguard (ACS) includes such costs as a countermeasure cost of purchase, development, licensing, a cost of annual maintenance and repair, etc. ACS is calculated using the formula below:
\[ \text{ACS} = \frac{C}{\$} \text{/year} \]

where \( C \) \( \$ \) – all costs associated with the buying, developing and further exploiting of the purchased safeguards;

- The value or benefit of a safeguard (cost/benefit) is calculated using the formula below:
  \[ \frac{C}{B} = ALE_{\text{pre}} - ALE_{\text{post}} - \text{ACS} \]
  where \( \frac{C}{B} \) – cost and/or benefit of a safeguard; \( ALE_{\text{pre}} \) – pre-countermeasure of the annualised loss expectancy; \( ALE_{\text{post}} \) – post-countermeasure of the annualised loss expectancy.

The pure quantitative analysis is not possible because some aspects cannot be quantified in money figures. Hence, there is the qualitative risk analysis which is more likely scenario based and is used to rank threats. The process of performing the qualitative risk analysis includes such factors as experience, judgment, and intuition, and so on.

Finally, SAVO is easily modifiable and extendable through adding additional subclasses to the core classes such as the class ‘Asset’ or ‘Vulnerability’.

### 4.3 The security algorithm-standard ontology

In this section, we introduce the security algorithm-standard ontology (SASO) which defines security algorithms and standards, security concepts, security credentials, assurance levels, and security objectives as ontology’s classes that are used as “building blocks” for the security function ontology (SFO) presented below. In other words, these classes link system’s security objectives and high-level security policies with low-level technical countermeasures. The interrelations among main classes of SASO and SFO are illustrated in Figure 4.6.

![Figure 4.6. Interrelations among SASO and SFO](image)

As previously mentioned, SAVO integrates SFO through the use of the class ‘SecurityFunction’ which assures security objectives (the class ‘SecurityObjective’ from SASO) specified by system managers or administrators using the security policies. It is worth mentioning that in this
context security objectives are used by defenders ("good guys") and should be distinguished from security objectives of attackers ("bad guys"). Hence, in this thesis we talk about security objectives from the defender’s perspective.

The main class of SFO is ‘SecurityFunction’ which has the property ‘hasSecParams’ that can have SASO classes ‘SecurityConcept’ and ‘SecurityAlgorithm’ as values. Since many security algorithms and concepts have various assurance levels and employ security credentials for access control, some of SASO classes including ‘SecurityConcept’ and ‘SecurityAlgorithm’ have properties ‘hasSecCredential’ and ‘hasSecAssurance’ which can have SASO classes ‘SecurityCredential’ and ‘SecurityAssurance’ as values. The classes ‘SecurityFunction’, ‘SecurityConcept’, and ‘SecurityAlgorithm’ assure various security objectives specified by the SASO class ‘SecurityObjective’.

### 4.3.1 The class SecurityObjective

In computer science, security objectives are goals that describe the ways to minimise risks and vulnerabilities applied to assets. For example, DES (or other symmetric cryptography algorithms) is declared to have confidentiality. Besides, security objectives allow a user to find algorithms, mechanisms, protocols, or policies based on the required security objectives. For example, a user can state, “find all instances of security algorithms that provide confidentiality” and receive a list of symmetric security algorithms. Or for example, a user does not know in advance what type of the security algorithm or concept she will use for protecting messages but she can specify that this algorithm should support integrity and authentication. Hence, when the system is ready to be deployed as a real system, the system manager can use SSL as one of the possible solutions.

![Figure 4.7. The class ‘SecurityObjective’](image)

The class ‘SecurityObjective’ is the main class and used to map high-level mission requirements to low-level component (service) implementations. It includes 10 subclasses, as illustrated in Figure 4.7. We demonstrate below how the security function ontology uses the class ‘SecurityObjective’ and its subclasses. The major security objectives are described as follows:
• ‘Confidentiality’ is concerned with the process of protecting data from unauthorised use or users;
• ‘Authorisation’ is concerned with the access to system resources such as data and operations;
• ‘Integrity’ is concerned with maintaining the consistency and correctness of information;
• ‘Availability’ is concerned with preventing the exhaustion of resources and the denial of services to authorised users;
• ‘Nonrepudiation’ is concerned with preventing an individual or entity from denying having performed an action such as sending or receiving information;
• ‘Authenticity’ is concerned with validating who the entity claims to be;
• ‘Auditability’ deals with issues such as detecting, recognising, recording, analysing security related activities;
• ‘Identification’ is the process by which a subject provides an identity;
• ‘Accountability’ forces subjects to be accountable for their actions;
• ‘Survivability’ deals with functioning during and after attacks.

4.3.2 The class SecurityAlgorithm

The class ‘SecurityAlgorithm’ is utilised to describe various security algorithms. Its structure is demonstrated in Figure 4.8 where the main class called ‘SecurityAlgorithm’ has three major subclasses including ‘SAlgEncryption’, ‘SAlgSignature’, and ‘SAlgKeyExchange’.

Figure 4.8. The class ‘SecurityAlgorithm’

The class ‘SAlgEncryption’ defines classes that deliver encryption/decryption capabilities. It consists of two subclasses (the classes ‘SAlgSymmetric’ and ‘SAlgAsymmetric’) that represent symmetric and asymmetric cryptographic algorithms. The class ‘SAlgKeyExchange’ specifies
cryptographic key exchange algorithms while the class ‘SAlgSignature’ includes classes that are responsible for digital signatures and cryptographic hash functions. All details regarding these algorithms are presented below.

For example, the AES cryptographic algorithm with the 256 bits key can be expressed in the following way:

\[
\text{Security.Algorithm.SAlgEncryption.SAlgSymetric.AES(256).}
\]

The Diffie-Hellman algorithm can be specified in the similar way:

\[
\text{Security.Algorithm.SAlgKeyExchange.DeffieHellman.}
\]

Cryptographic algorithms, as illustrated below, can have additional properties such as the version, the block size, the key size and type (symmetric, asymmetric private or public), security objectives they are used to achieve, and security mechanisms, policies, protocols, and concepts they are applied to.

**Encryption algorithms**

**Symmetric cryptography algorithms**

Symmetric cryptography algorithms are defined by the class ‘SAlgSymmetric’, as illustrated in Figure 4.9, which has several subclasses including DES (the class ‘DES’), Triple-DES (the class ‘3DES’), AES (the class ‘AES’), Blowfish (the class ‘Blowfish’), Twofish (the class ‘Twofish’), IDEA (the class ‘IDEA’), CAST (the class ‘CAST’), Skipjack (the class ‘Skipjack’), RC2 (the class ‘RC2’), RC4 (the class ‘RC4’), RC5 (the class ‘RC5’), and RC6 (the class ‘RC6’ that are introduced in details in Appendix A.

![Figure 4.9. The class ‘SAlgSymmetric’](image)

Data Encryption Standard (DES) is defined by the class ‘DES’ and is employed as a parameter for SFO security functions (described below). For example, the functions, which state that PGP
and the DES algorithm are utilised and the length of a symmetric key equals 56 bits, can be specified as follows:

\[
\text{KeyAlgorithm(SymmetricKey,DES,PGP).}
\]
\[
\text{KeySize(SymmetricKey,56).}
\]

**Asymmetric cryptography algorithms**

In the asymmetric cryptography algorithms messages are encrypted with the public key and decrypted with the private key, while the symmetric cryptography uses the same key for both encryption and decryption (represented by the class ‘SAlgAsymmetric’). An example of the asymmetric key algorithm is Rivest-Shamir-Adelman (RSA) encryption, which is the public key cryptosystem (see Figure 4.8). It can be broken by brute force attacks, mathematical attacks and timing attacks. For instance, the fact that the cryptographic key is distributed using RSA and IKE algorithms can be stated as follows:

\[
\text{KeyDistribution(K[C2,C1],IKE,RSA).}
\]

Majority of public-key cryptosystems (RSA) use either integer or polynomial arithmetic with very large numbers/polynomials. It imposes a significant load in storing and processing keys and messages. But there are some alternatives and one of the promising one is Elliptic Curve Cryptography (ECC) which offers same security with smaller key length. For example, if the RSA key size is 1088 bits then the ECC key length should be equal 160 bits in order to provide same protection.

**Digital signature algorithms**

Digital signatures allow to verify the author, date and time of signatures, authenticate message contents, and be verified by third parties. The main supported security goal of digital signatures is integrity. They are represented by the class ‘SAlgSignature’, as illustrated in Figure 4.10.

---

**Figure 4.10. The class ‘SAlgSignature’**

---

70
RSA and hash functions such as CBCMAC (the class ‘CBCMAC’) which is based on a block cipher, HMAC (the class ‘HMAC’) which is constructed from two hash functions, MD5 (the class ‘MD5’) that is a public hash algorithm used to create checksums in order to ensure data integrity which has been broken recently, SHA-1 (the class ‘SHA1’) that is a public hash algorithm employed to create checksums in order to ensure data integrity and also has been cracked recently, and SHA-256 (the class ‘SHA256’) which is similar to SHA-1 but has larger keys (supposed to be broken in the nearest future) and can be used for creating digital signatures.

For example, the fact that an email system supports PGP and employs SHA-256 for signing emails can be expressed in the following way where the size of the private key equals 256:

\[
\text{KeyAlgorithm(PrivateKey,SHA256,PGP).} \\
\text{KeySize(PrivateKey,256).} \\
\text{Signed(email,PrivateKey).}
\]

**Cryptographic key exchange algorithms**

As previously mentioned, key delivery is a difficult task. Currently, there are several key exchange protocols, as illustrated in Figure 4.8:

- The Deffie-Hellman protocol (the class ‘DeffieHellman’) allows two parties that have no prior knowledge of each other to jointly establish a shared private key (used to encrypt subsequent communications using symmetric cryptography) over an insecure communications channel. For example, the fact that the cryptographic key is distributed using D-H and IKE algorithms can be stated as follows:

\[
\text{KeyDistribution(K[C2,C1],IKE,DeffieHellman)}
\]

Examples for other key algorithms are similar.

- The key exchange algorithm (KEA) (the class ‘KEA’) is based upon the Diffie-Hellman algorithm and utilises the symmetric cryptography algorithm called Skipjack made public recently by NSA.
- The Internet key exchange (IKE) algorithm (the class ‘IKE’) uses the Diffie-Hellman key exchange to set up shared session secrets and deliver cryptographic keys. It is used in the IPsec protocol and builds upon the Oakley protocol.
- The Oakley key agreement protocol (the class ‘Oakley’) allows authenticated parties to exchange cryptographic keys across insecure channels utilising the Deffie-Hellman algorithm.
4.3.3 The class SecurityConcept

The class ‘SecurityConcept’ consists of three subclasses, as shown in Figure 4.11, including ‘SConMechanism’, ‘SConProtocol’ and ‘SConSecurityPolicy’. We inherit the similar structure as in the class ‘SecurityConcept’ of the main security ontology from [KLK05]. The difference between security protocols and security mechanisms is the following: security protocols (the class ‘SConProtocol’) specify how to fulfill certain tasks using certain steps while security mechanisms (the class ‘SConMechanism’) are implementations of security protocols. Security policies (the class ‘SConSecurityPolicy’) define rules for access control models.

![Figure 4.11. The class ‘SecurityConcept’](image)

The class ‘SConMechanism’ mainly defines various types of firewalls, proxies and virtual machines (VMs). The class ‘SConProtocol’ has many subclasses that specify different security protocols used in authentication (the class ‘SConAuthentication’), encryption (the class ‘SConEncryption’), key management (the class ‘SConKeyMngmt’), digital signatures (the class ‘SConSignature’), secure e-mail systems (the class ‘SConEmail’), and secure communications (the class ‘SConSecureCommunication’). The class ‘SConSecurityPolicy’ just describes rules and policies used in access control models that support authorisation.

**Security protocols**

In this section, we present the class ‘SConProtocol’ and its subclasses as illustrated in Figure 4.12.

**Secure communications**

Secure communications are defined by the class ‘SConSecureCommunication’, as shown above in Figure 4.12, which consists of a number of subclasses (security protocols) including ‘SSL’, ‘SSH’, ‘TLS’, ‘HTTPS’, ‘SHTTP’, ‘SET’, ‘IPSec’, ‘PPTP’, ‘L2TP’, and ‘L2F’. More detailed definitions of these protocols can be found in Appendix A and [Wiki07] or [STC05].

For example, the fact that the trusted channel between two parties Alice and Bob is organised using SSL can be express as follows:

```
TrustedChannel(Alice, Bob, SSL).
```
Figure 4.12. The class ‘SConProtocol’

(Email security protocols)

In this section, we introduce a few protocols and standards for securing e-mails.

S/MIME (Secure / Multipurpose Internet Mail Extensions) (the class ‘SMIME’) is a standard that allows to sign e-mails using public key encryption. It requires a certificate from a trusted third CA (Certification Authority) to protect e-mails and attachments and provide message encryption using DES, 3DES, and RC2. Delivered security goals include authentication, integrity, non-repudiation, confidentiality, and identity proofing.

PGP (Pretty Good Privacy) (the class ‘PGP’) is a computer program mainly used to encrypt emails. It utilises IDEA, CAST, and 3DES algorithms for encrypting messages and RSA for digital signatures, key distribution, and key management. For example, the fact that the symmetric key is generated if SMIME and DES with the key length of 56 bits are used can be expressed as follows:

KeyGenerated(SymmetricKey)← KeyAlgorithm(SymmetricKey,DES,SMIME),KeySize(SymmetricKey,56).

(Authentication protocols)

The Kerberos (the class ‘Kerberos’) protocol is an authentication protocol that allows distributed parties to communicate securely over an insecure network. It uses symmetric key cryptography and a trusted third party called KDC (Key Distribution Centre). Kerberos enables single sign-on (SSO) and protection of key exchange mechanisms, however, it has several points of failure including:

- KDC is a single point of failure because it should handle many requests;
- Tickets may be compromised because they are temporary stored on users’ computers;
• Initial authentication is rather vulnerable to various password guessing attacks;
• Kerberos does not deliver confidentiality and protection of network traffic;
• Changes of a user’s password also change a secret key, and consequently, the KDC
database needs to be updated as well.

Regarding to the CIA triad, Kerberos delivers only authentication and integrity.

Another authentication protocol is CHAP (Challenge Handshake Authentication Protocol) (the
class ‘CHAP’) that is used to encrypt passwords during a logon process. It utilises a
challenge/response method of authentication where credentials are hashed using MD5. CHAP
verifies periodically the identity of a client using a three way handshake. CHAP provides
authentication only.

The last protocol, we introduce here, is SAML (Security Assertion Markup Language) (the class
‘SAML’). It is mainly used in the Web services context for authentication and authorisation of
data between security domains. The main problem it solves is a SSO (single sign-on) problem.
Hence, SAML has become a standard underlying many SSO solutions.

(Encryption, key management, and signature protocols)

In this section, we describe protocols that are mainly related to XML data and used in the Web
services context.

XML encryption (the class ‘XMLEnc’) is a specification that defines the steps of encrypting and
decrypting data. It also introduces the XML syntax for representing encrypted data and specifies
a list of encryption algorithms including 3DES, AES, and RSA.

XKMS (XML Key Management Specification) (the class ‘XKMS’) is a protocol which
describes the distribution and registration of public keys in order to provide encryption and
authentication. XML digital signatures (the class ‘XMLDSig’) are digital signatures created for
use in the XML context (XML data).

Security mechanisms

In this section, we introduce the class ‘SConMechanism’ and its subclasses, as illustrated in
Figure 4.13.

VPN (Virtual private network) (the class ‘VPN’) is used to secure communications over un-
trusted networks using tunnelling protocols that wrap and protect network packets.

OnionRouter (the class ‘OnionRouter’) also called onion routing is a method for anonymous
communication over computer networks. However, it has several drawbacks including as a lack
of defence against timing analysis attacks, intersection attacks, and predecessor attacks.
Currently, there is the second-generation onion router called Tor.
A firewall is a mechanism of preventing unauthorized access to private networks. It is usually used to reduce ability of external intruders to affect internal networks, to block access to some sites or services, and monitor and audit communications.

An intrusion detection system (IDS) [Axe98,KLC+04,JWZ03] is a second line of defence and usually a complimentary to a firewall. Firewalls and IDSs can be network-based (the class ‘NetBasedFirewallIDS’), host-based (the class ‘HostBasedFirewallIDS’), application-based (the class ‘AppBasedFirewallIDS’ and the class ‘AppProxy’), and XML/SOAP firewalls (the class ‘SOAPFirewall’ which are specially designed to address security issues of Web services and which can have software or hardware implementations.

The last class, we introduce here, is VM (Virtual Machine) (the class ‘VM’) which is related to software that creates a virtual environment between a computer platform and its operating system in order to allow users to launch and operate software on this virtual (abstract) machine. The class ‘MACJVM’ defines a Java virtual machine with additional security constraints and access control mechanisms.

Security policies
In this section we introduce security policies (the class ‘SConSecurityPolicy’) that are represented by access control models, as shown in Figure 4.14.

After a user (subject) has been identified and authenticated and been accountable, he/she should be authorised for getting access to certain resources and performing certain actions. Authorisation is provided by systems through access control mechanisms. There are two major types of access controls (the class ‘SConSecurityPolicy’): for the use in commercial companies (the class ‘CommercialPolicy’) and in military (the class ‘MilitaryPolicy’). The class ‘CommercialPolicy’ includes discretion access control (DAC) (the class ‘DAC’), nondiscretion access control (the class ‘SConNonDAC’), domain policies (the class ‘DomainPolicy’), the Chinese Wall model (the class ‘ChineseWall’), and the Clark-Wilson model (the class ‘ClarkWilson’).
In DAC access is granted or denied to a subject in a discretionary environment based on the subject’s identity. DAC is usually implemented on objects using access control lists (ACLs) that specify types of access for subjects or group of subjects. DAC is usually used in commercial systems.

The class ‘SConNonDAC’ specifies access controls usually used in rule-based systems which applies a set of rules, constraints, and filters to grant or restrict access to certain users on certain subjects or performing certain actions. Role-based access control (RBAC) (the class ‘RBAC’), which is a subclass of the class ‘SConNonDAC’, allows subjects to access objects through the use of subject roles RBAC is usually utilised in commercial systems.

The class ‘ChineseWall’ and ‘ClarkWilson’ specify access control models and are usually utilised in commercial application.

The class ‘SConMAC’ defines the class of mandatory access controls (MAC) that relies on usage of classification labels (subjects are labelled by their level of clearance while objects are labelled by their level of sensitivity). MAC is considered more secure than DAC but less scalable and flexible and is usually utilised in military systems. Other models used in military are TBAC and BLP (Bell-LaPadula Model) (the class ‘BLP’).

Task-based access control (TBAC) is defined by the class ‘TBAC’ and allows subjects to access objects through the use of subject tasks or work functions.

The class ‘MACJVMPolicy’ defines security policy for Java virtual machines.

4.3.4 The class SecurityAssurance

The class ‘SecurityAssurance’ specifies assurance methods for security algorithms, protocols, and mechanisms. Currently, it has only one subclass called ‘CommonCriteria’, as illustrated in Figure 4.15. Since our ontology is extensible, new classes such as TCSEC (Trusted Computer
System Evaluation Criteria) or ITSEC (Information Technology Security Evaluation Criteria) can be added easily.

The Common Criteria (CC) [CC06] is an international standard for a computer security evaluation which is used to evaluate security measures needed for certain software applications. The CC consists of three parts: Introduction and General Model, Functional Requirements, Assurance Requirements.

Part 2 (Functional Requirements) describes functional requirements. These requirements introduce the desired security behaviour expected of a Target of Evaluation (TOE) and specify security properties that users can detect by direct interaction with the TOE. There are 11 Security Functional Classes: 1) Class FAU (security audit); 2) Class FCO (communication); 3) Class FCS (cryptographic support); 4) Class FDP (user data protection); 5) Class FIA (identification and authentication); 6) Class FMT (security management); 7) Class FPR (privacy); 8) Class FPT (protection of the trusted security functions); 9) Class FRU (resource utilization); 10) Class FTA (TOE access); and 11) Class FTP (trusted path/channels).

Part 3 (Assurance Requirements) covers assurance requirements for TOEs several areas such as life cycle support, vulnerability assessments, and the complete range of security assurance checks. Seven evaluation assurance levels (EALs) describe how systems are designed, checked, and tested:

- ‘EAL1’ – Functionally tested;
- ‘EAL2’ – Structurally tested;
- ‘EAL3’ – Methodically tested and checked;
- ‘EAL4’ – Methodically designed, tested, and reviewed;
- ‘EAL5’ – Semi-formally designed and tested;
- ‘EAL6’ – Semi-formally verified, designed, and tested;
- ‘EAL7’ – Formally verified, designed, and tested.
However, the CC is rather generic and does not provide methodology for security measurements. It does not guarantee security of data after users’ actions and does not address administrative issues outside the specific security scope.

### 4.3.5 The class SecurityCredential

The class ‘SecurityCredential’, illustrated in Figure 4.16, is used for identification and authentication purposes.

![Figure 4.16. The class ‘SecurityCredential’](image)

It has subclasses including ‘Certificate’ with subclass ‘X509Certificate’, ‘Cookie’, ‘Domain’, ‘IPAddress’, ‘DigSignature’, ‘PrivateKey’, ‘Password’, and ‘OnetimePassword’. We partially adopt the class structure from [KLK05] by taking into account electronic tokens (the class ‘ElectronicToken’).

### 4.4 The security function ontology

In this section, we introduce the security function ontology (SFO) which is used together with the security algorithm-standard ontology (SASO) in order to provide vocabulary. The classes ‘SecurityAlgorithm’ and ‘SecurityConcept’ can be used as security function parameters as well as for analysing and reasoning these functions. For example, the function ‘Signed’ assures several security objectives including integrity, authenticity, and non-repudiation. These security functions are extracted from [Kha05], modified to provide a more structured view, and utilised for specifying dynamic security properties of components and services, while the original security functions can be employed only for expressing static security properties. Besides, we develop SFO using these security functions. The dependencies among various SFO and SASO classes have been illustrated in the previous section. It is worth mentioning that Sections 4.4.3-4.4.8 are organised in the following way: first, we briefly describe the security function class and its constituent security functions; then, we demonstrate how these functions can be
specified; and finally, we present short examples of the usage of these security functions. It is necessary to mention that the detailed description of the security functions introduced below can be found in [Kha05] and Appendix A.

### 4.4.1 The class Properties

The main class is ‘Prop’ (properties) with subclasses ‘NonFuncProp’, which describes non-functional properties such as security (‘SecProp’) or performance, and ‘FuncProp’, which define functional properties such as structural adaptation and reconfiguration, as illustrated in Figure 4.17.

![Figure 4.17. The class ‘Prop’](image)

‘SecProp’ describes required and ensured security properties (utilized as values of security functions). A component or service interface may have the property ‘hasPropValue’ which has the class ‘Prop’ as a value.

### 4.4.2 Classes of security functions

There are six classes of the security functions (the class ‘SecurityFunction’) extracted from [Kha05] that are derived from merging 11 Common Criteria classes including ‘PrivacyFunc’, ‘CryptoSupportFunc’, ‘IdentificationAuthorisationFunc’, ‘UserDataProtectionFunc’, ‘TrustedChannelFunc’, and ‘SecurityAuditResourceUtilisationFunc’. Moreover, we develop two additional classes including ‘TimeFunc’ and ‘ProbabilityFunc’ used in conjunction with security functions to specify dynamic and unpredictable nature of security in component (service) based software systems, as illustrated in Figure 4.18.

![Figure 4.18. The class ‘SecurityFunction’](image)

Each of these classes has subclasses with the property ‘hasSecurityObjective’ and which are mapped on the security functions with parameters from the classes ‘SecurityAlgorithm’,
‘SecurityConcept’, and security objectives (‘SecurityObjective’), as described in the previous sections. Also, we modified the names of the security functions from [Kha05] in order to make them more intuitively understandable and organised them in the classes described above. However, we did not include the security function ‘MgmtSecurityAttribute(Q,O,X)’ of the class ‘IdentificationAuthorisationFunc’ used to perform the security operation ‘O’ on the object ‘X’ because we consider that the security function ‘Authenticated(X,Q,O)’ from the same class fully fulfills our needs. The difference between two functions is that in the former the parameter ‘Q’ is authenticated to perform the operation ‘O’ which can be not security-related while in the latter ‘Q’ is permitted to perform only security operations. Besides, we added the additional security function to the class ‘UserDataProtectionFunc’ called ‘ObjectExchanged(P,Q,X)’ that allows the object ‘X’ to be exchanged securely between two parties, ‘P’ and ‘Q’.

4.4.3 The class of security functions for cryptographic support

The class of security functions for cryptographic support (the class ‘CryptoSupportFunc’), as illustrated in Table 4.1 and Figure 4.19, is designed to support encryption/decryption algorithms. It consists of two families: cryptographic key management, which is used to address the management aspects of cryptographic keys, and cryptographic operations, which concerns with the operational use of the cryptographic keys. Security functions or their sets assure certain security goals such as confidentiality, integrity, availability, etc, introduced in the previous sections. For example, the function Signed(X,K^{-1}) assures authenticity, integrity, and non-repudiation, as depicted in Figure 4.19.

Figure 4.19. The class ‘CryptoSupportFunc’
<table>
<thead>
<tr>
<th>No</th>
<th>Security function name</th>
<th>Description</th>
<th>Security goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shared(K,P,Q)</td>
<td>The shared (symmetric cryptography) key ‘K’ is shared between the entities ‘P’ and ‘Q’ (e.g., the users Alice and Bob).</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>2</td>
<td>PairOf(K’,K)</td>
<td>Asymmetric cryptography public (‘K’) and private (‘K’) keys are used in pairs.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>3</td>
<td>Owned(K,P)</td>
<td>The key ‘K’ is owned by the entity ‘P’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>4</td>
<td>KeyAlgorithm(K,A,S)</td>
<td>The key ‘K’ employs the algorithm ‘A’ in conjunction with the standard ‘S’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>5</td>
<td>Encrypted(X,K)</td>
<td>The object ‘X’ (e.g., a message or a file) is encrypted with any type of the key (shared or public) ‘K’ by the undefined entity.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>6</td>
<td>KeySize(K,L)</td>
<td>The size of the key ‘K’ equals ‘L’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>7</td>
<td>KeyGenerated(K)</td>
<td>The key ‘K’ is generated.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>8</td>
<td>KeyDistribution(K,A,S)</td>
<td>The key ‘K’ is distributed using the distribution algorithm ‘A’ that is supported by the standard (or protocol) ‘S’.</td>
<td>Confidentiality, Integrity.</td>
</tr>
<tr>
<td>9</td>
<td>Signed(X,K-1)</td>
<td>The object ‘X’ is digitally signed with the private key ‘K-1’.</td>
<td>Authenticity, Integrity, Non-repudiation.</td>
</tr>
<tr>
<td>10</td>
<td>BelievesProduced(P,Q,X)</td>
<td>The entity ‘P’ believes that the entity ‘Q’ once produced the object ‘X’ in the past.</td>
<td>Authenticity, Non-repudiation.</td>
</tr>
<tr>
<td>11</td>
<td>BelievesShared(P,Q,X)</td>
<td>The entity ‘P’ believes that it shares the object ‘X’ with another entity ‘Q’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>12</td>
<td>KeyExchanged(K,P,Q)</td>
<td>The key ‘K’ is security exchanged between the entities ‘P’ and ‘Q’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>13</td>
<td>KeyOperation(K,O)</td>
<td>The operation ‘O’ can be performed with the key ‘K’.</td>
<td>Availability.</td>
</tr>
<tr>
<td>14</td>
<td>ObjectAuthenticated(Q,X,P)</td>
<td>The object ‘X’ is authenticated by the entity ‘Q’ to the entity ‘P’.</td>
<td>Integrity, Authenticity.</td>
</tr>
</tbody>
</table>

Table 4.1. Security functions for cryptographic support

Examples

<table>
<thead>
<tr>
<th>No</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shared(Key,Alice,Bob) - the symmetric key ‘Key’ is shared between the users ‘Alice’ and ‘Bob’.</td>
</tr>
<tr>
<td>2</td>
<td>PairOf(PrivateKey,PublicKey) – the pair of the private ‘PrivateKey’ and public ‘PublicKey’ keys can be used by the RSA asymmetric algorithm.</td>
</tr>
<tr>
<td>3</td>
<td>Owned(Key,Alice) – the cryptographic key ‘Key’ is owned by the user ‘Alice’.</td>
</tr>
<tr>
<td>4</td>
<td>KeyAlgorithm(Key,AES,IPSec) – the symmetric key ‘Key’ is used by the AES and IPSec security algorithms.</td>
</tr>
<tr>
<td>5</td>
<td>Encrypted(Email,Key) – the email ‘Email’ is encrypted with the key ‘Key’.</td>
</tr>
<tr>
<td>6</td>
<td>KeySize(Key,256) – the size of the key equals 256 bits.</td>
</tr>
<tr>
<td>7</td>
<td>KeyGenerated(Key) – the key ‘Key’ is generated.</td>
</tr>
<tr>
<td>8</td>
<td>KeyDistribution(Key[Alice,Bob],IKE,DiffieHellman) – the shared key ‘Key’ is distributed between ‘Alice’ and ‘Bob’ using IKE and Diffie-Hellman algorithms.</td>
</tr>
<tr>
<td>9</td>
<td>Signed(Email,PrivateKey) – the email ‘Email’ is signed using the private key ‘PrivateKey’.</td>
</tr>
<tr>
<td>10</td>
<td>BelievesProduced(Alice,Bob,Sig) – ‘Alice’ believes that ‘Bob’ once produced the signature ‘Sig’.</td>
</tr>
<tr>
<td>11</td>
<td>BelievesShared(Alice,Bob,EmountX) – ‘Alice’ believes she shares ‘EmountX’ with ‘Bob’.</td>
</tr>
<tr>
<td>12</td>
<td>KeyExchanged(Key,Alice,Bob) – the key ‘Key’ is exchanged between ‘Alice’ and ‘Bob’.</td>
</tr>
<tr>
<td>13</td>
<td>KeyOperation(Key,Sig) – the key ‘Key’ is employed to create the digital signature ‘Sig’.</td>
</tr>
<tr>
<td>14</td>
<td>ObjectAuthenticated(Alice,X,Bob) – ‘Alice’ authenticates the object ‘X’ to ‘Bob’.</td>
</tr>
</tbody>
</table>
The class of security functions for user data protection

The class of security functions for user data protection (the class ‘UserDataProtectionFunc’), as shown in Figure 4.20, is used to protect user data during its import, export, and storage.

**Figure 4.20. The class ‘UserDataProtectionFunc’**

<table>
<thead>
<tr>
<th>No</th>
<th>Security function name</th>
<th>Description</th>
<th>Security goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Known(X,P).</td>
<td>The object ‘X’ (trusted or un-trusted) is known to the entity ‘P’.</td>
<td>Availability.</td>
</tr>
<tr>
<td>16</td>
<td>Combined(X,Y).</td>
<td>The object ‘X’ (e.g., login) is combined with the object ‘Y’ (e.g., password).</td>
<td>Integrity.</td>
</tr>
<tr>
<td>17</td>
<td>Fresh(X).</td>
<td>The object ‘X’ is fresh, i.e., it was not previously processed or transmitted by any entity.</td>
<td>Integrity.</td>
</tr>
<tr>
<td>18</td>
<td>Controls(P,X).</td>
<td>The entity ‘P’ fully controls the object ‘X’, i.e., other entities cannot change or control the value of ‘X’.</td>
<td>Authenticity.</td>
</tr>
<tr>
<td>19</td>
<td>Sees(P,X).</td>
<td>The entity ‘P’ receives and sees the object ‘X’.</td>
<td>Availability.</td>
</tr>
<tr>
<td>20</td>
<td>BelievesSees(P,Q,X).</td>
<td>The entity ‘P’ believes that the entity ‘Q’ has received the object ‘X’ at some point in the past.</td>
<td>Non-repudiation, Availability.</td>
</tr>
<tr>
<td>21</td>
<td>PartsOf(X,Y).</td>
<td>The object ‘X’ is a part of another object ‘Y’.</td>
<td>Integrity.</td>
</tr>
<tr>
<td>22</td>
<td>Protected(X,P,Q).</td>
<td>The object ‘X’ is securely protected while transferred between the entities ‘P’ and ‘Q’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>23</td>
<td>ObjectExchanged(P,Q,X).</td>
<td>The object ‘X’ is exchanged between the entities ‘P’ and ‘Q’.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>24</td>
<td>ObjectExchangedIntegrity(P,Q,X).</td>
<td>The integrity of the object ‘X’ is maintained during exchange between</td>
<td>Integrity, Confidentiality.</td>
</tr>
</tbody>
</table>
Table 4.2. Security functions for user data protection

<table>
<thead>
<tr>
<th>No</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Known(Amount, Alice)</td>
<td>‘Alice’ knows the value of ‘Amount’, however, she may not believe that the value is true.</td>
</tr>
<tr>
<td>16</td>
<td>Combined(LoginOfAlice, Pass)</td>
<td>the login of ‘Alice’ ‘LoginOfAlice’ is combined with her password ‘Pass’.</td>
</tr>
<tr>
<td>17</td>
<td>Fresh(TaxReturn)</td>
<td>the tax return ‘TaxReturn’ can be send by the tax office only once in the same financial year.</td>
</tr>
<tr>
<td>18</td>
<td>Controls(Alice, Pass)</td>
<td>‘Alice’ fully controls her own password.</td>
</tr>
<tr>
<td>19</td>
<td>Sees(Alice, Pin)</td>
<td>‘Alice’ can see the value of ‘Pin’.</td>
</tr>
<tr>
<td>20</td>
<td>BelievesSees(Alice, Bob, Pin)</td>
<td>‘Alice’ believes that ‘Bob’ has received ‘Pin’.</td>
</tr>
<tr>
<td>21</td>
<td>PartsOf(AuthInfo, Pass)</td>
<td>the password ‘Pass’ is the part of authentication information ‘AuthInfo’.</td>
</tr>
<tr>
<td>22</td>
<td>Protected(SymKey, Alice, Bob)</td>
<td>the symmetric key ‘SymKey’ is protected during its transfer between ‘Alice’ and ‘Bob’.</td>
</tr>
<tr>
<td>23</td>
<td>ObjectExchanged(Alice, Bob, PubKey)</td>
<td>the public key ‘PubKey’ is exchanged between ‘Alice’ and ‘Bob’.</td>
</tr>
<tr>
<td>24</td>
<td>ObjectExchangedIntegrity(Alice, Bob, Email)</td>
<td>integrity of the object ‘Email’ exchanged between ‘Alice’ and ‘Bob’ is maintained.</td>
</tr>
<tr>
<td>25</td>
<td>ImportedObject(Email, Site)</td>
<td>the object ‘Email’ is imported from the site ‘Site’.</td>
</tr>
<tr>
<td>26</td>
<td>ExportedObject(CardNumber, Site)</td>
<td>the object ‘CardNumber’ is exported from the site ‘Site’.</td>
</tr>
<tr>
<td>27</td>
<td>InternalTransfer(KeyValue, Proc1, Proc2)</td>
<td>the object ‘KeyValue’ is protected during its transfer between two system process: ‘Proc1’ and ‘Proc2’.</td>
</tr>
<tr>
<td>28</td>
<td>RollBack(Payment, Cancel, Paid, Unpaid)</td>
<td>in case of canceling ‘Cancel’ the state of payment ‘Payment’ rolls back from ‘Paid’ to ‘Unpaid’.</td>
</tr>
<tr>
<td>29</td>
<td>ResidualProtection(Amount)</td>
<td>the object ‘Amount’ is protected if it is logically deleted and entities cannot see it after deleting.</td>
</tr>
<tr>
<td>30</td>
<td>StorageIntegrity(Amount)</td>
<td>storage integrity the object ‘Amount’ is provided.</td>
</tr>
</tbody>
</table>
4.4.5 The class of security functions for identification & authorisation

The class of security functions for identification and authorisation (the class ‘IdentificationAuthorisationFunc’), as depicted below in Figure 4.21, is used to address requirements for security functions to establish and check identities claimed by users.

![Figure 4.21. The class ‘IdentificationAuthorisationFunc’](image)

<table>
<thead>
<tr>
<th>No</th>
<th>Security function name</th>
<th>Description</th>
<th>Security goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>SeesUserAttribute(P,Q).</td>
<td>The entity ‘P’ sees the user attributes including the identity details, the origin site, and the role of the entity ‘Q’.</td>
<td>Availability, Authenticity.</td>
</tr>
<tr>
<td>33</td>
<td>Acl(P,R,X,O,C).</td>
<td>The entity ‘P’ with the role ‘R’ can perform the operation ‘O’ on the object ‘X’ under the optional condition ‘C’.</td>
<td>Availability, Authorisation.</td>
</tr>
<tr>
<td>34</td>
<td>Authenticated(Q,X,O).</td>
<td>The entity ‘Q’ is allowed to perform the operation ‘O’ on the object ‘X’.</td>
<td>Authenticity, Authorisation.</td>
</tr>
<tr>
<td>35</td>
<td>SeesAuthenticateFail(P,U,Q).</td>
<td>The action ‘U’ is taken by the entity ‘P’ for authentication failure of the entity ‘Q’, i.e., this security function can be used to monitor the number of authentication failures and the actions taken by entities.</td>
<td>Authenticity, Authorisation.</td>
</tr>
</tbody>
</table>

Table 4.3. Security functions for identification & authorisation

Examples

<table>
<thead>
<tr>
<th>No</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>SeesUserAttribute(Alice,Bob) – ‘Alice’ can see the user attributes of ‘Bob’.</td>
</tr>
<tr>
<td>33</td>
<td>Acl(Alice,Admin,PassFile,Read,No) – ‘Alice’ who is a system administrator ‘Admin’ can perform the operation ‘Read’ on the password file ‘PassFile’ without any conditions.</td>
</tr>
<tr>
<td>34</td>
<td>Authenticated(Alice,PassFile,Read) – ‘Alice’ is allowed to perform the operation read on the object ‘PassFile’.</td>
</tr>
<tr>
<td>35</td>
<td>SeesAuthenticateFail(Bob,BlockAccess,Alice) – ‘Alice’ failed to authenticate herself to ‘Bob’, hence, her access is blocked ‘BlockAccess’.</td>
</tr>
</tbody>
</table>

4.4.6 The class of security functions for security audit and resource utilisation

The class of security functions for security audit and resource utilisation (the class ‘SecurityAuditResourceUtilFunc’), as illustrated in Figure 4.22, is regarded more as a security service. It is used for detecting, analysing, and storing security-related activities. The resulting
information can be used for auditing and identifying what kind of security activities occur and caused by whom.

![Figure 4.22. The class ‘SecurityAuditResourceUtilFunc’](image)

<table>
<thead>
<tr>
<th>No</th>
<th>Security function name</th>
<th>Description</th>
<th>Security goal</th>
</tr>
</thead>
</table>

**Table 4.4. Security functions for security audit and resource utilisation**

**Examples**

<table>
<thead>
<tr>
<th>No</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>AuditResponse(Alice,CloseSession,ReadPassFile) – ‘Alice’ closes a session with any unauthorized entity if this entity tries to read the password file.</td>
</tr>
<tr>
<td>37</td>
<td>GeneratesAudit(Alice,Bob,Login) – ‘Alice’ generates the report that ‘Bob’ logged into the system.</td>
</tr>
<tr>
<td>38</td>
<td>AnalysesSecurity(Alice,SignatureBased,BobFile) – ‘Alice’ employs the signature-based detection method ‘SignatureBased’ on the security event file ‘BobFile’ of ‘Bob’.</td>
</tr>
<tr>
<td>39</td>
<td>SelectsAudit(Alice,UnsuccessfulLogin,BobFile,Month) – ‘Alice’ can choose all unsuccessful login attempts ‘UnsuccessfulLogin’ of ‘Bob’ in a particular month ‘Month’ from his security event file ‘BobFile’.</td>
</tr>
</tbody>
</table>

### 4.4.7 The class of security functions for privacy

The class of security functions for privacy (the class ‘PrivacyFunc’) concerns privacy requirements of users (Figure 4.23).

![Figure 4.23. The class ‘PrivacyFunc’](image)
Table 4.5. Security functions for privacy

<table>
<thead>
<tr>
<th>No</th>
<th>Security function name</th>
<th>Description</th>
<th>Security goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>BelievesAnonymity(P,O,X).</td>
<td>The identity of the entity 'P' during the operation 'O' on the object 'X' is not known to any other entities.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>41</td>
<td>Unlinkability(P,O,X).</td>
<td>The entity 'P' cannot be linked with the operation 'O' performed on the object 'X' by any other entities.</td>
<td>Confidentiality.</td>
</tr>
<tr>
<td>42</td>
<td>Unobservability(P,O,X).</td>
<td>Other entities cannot observe the progress of the operation 'O' on the object 'X' performed by the entity 'P'.</td>
<td>Confidentiality.</td>
</tr>
</tbody>
</table>

Examples

<table>
<thead>
<tr>
<th>No</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>BelievesAnonymity(Alice,Credit,CardNumber) – 'Alice' believes that her transaction 'Credit' with the credit card details 'CardNumber' stays anonymous.</td>
</tr>
<tr>
<td>41</td>
<td>Unlinkability(Alice,Debit,CardNumber) – the transaction 'Debit' with the credit card details 'CardNumber' of 'Alice' cannot be linked with 'Alice' by any other user.</td>
</tr>
<tr>
<td>42</td>
<td>Unobservability (Alice,Credit,CardNumber) – other users cannot observe that 'Alice' does the transaction 'Credit' with the credit card number 'CardNumber'.</td>
</tr>
</tbody>
</table>

4.4.8 The class of security functions for trusted channel

This class is of security functions for trusted channels (the class ‘TrustedChannelFunc’) used to protect communications between users or other entities from modification or disclosure (Figure 4.24).

Figure 4.24. The class ‘TrustedChannelFunc’

<table>
<thead>
<tr>
<th>No</th>
<th>Security function name</th>
<th>Description</th>
<th>Security goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>TrustedChannel(P,Q,M).</td>
<td>The trusted method ‘M’ is used to organise the trusted channel between the entities ‘P’ and ‘Q’.</td>
<td>Confidentiality, Integrity.</td>
</tr>
<tr>
<td>44</td>
<td>TrustedPath(P,Q).</td>
<td>The trusted path is established between the entities ‘P’ and ‘Q’.</td>
<td>Confidentiality, Integrity.</td>
</tr>
</tbody>
</table>

Table 4.6. Security functions for trusted channel

Examples

<table>
<thead>
<tr>
<th>No</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>TrustedChannel(P,Q,M) – the trusted channel between ‘Alice’ and ‘Bob’ is organised using the SSL protocol.</td>
</tr>
<tr>
<td>44</td>
<td>TrustedPath(P,Q) – the trusted path is established between ‘Alice’ and ‘Bob’.</td>
</tr>
</tbody>
</table>
4.4.9 Time and probability

In this section, we describe the classes `TimeFunc` and `ProbabilityFunc` that are used as parameters by the class `SecurityFunction` described above. They allow to describe dynamic and unpredictable nature of distributed systems, component-based software systems in particular. Furthermore, these classes are widely used for specifying security properties that are time and event dependent and probabilistic.

**Time**

The class `TimeFunc` defines time. In traditional security, time is used to check expiration date of security certificates or licenses, security sessions, etc. Time is applied to express dynamic security properties and qualify their validity. We adapt coordinated universal time (UTC) as a reference point, which is measured from 1 January 1970. Normal day and time are used implying their translation to UTC.

**(Time points)**

The time point `<T>` describes exactly when security properties of components are valid. Time points are also used to express time intervals and time sequences. For example, Alice and Bob know that the shared key is valid at 9am but there are no guarantees that it will be valid at other time periods. This fact can be expressed in the following way:

```
Shared(key,Alice,Bob),Time(<9am>).
```

**(Time intervals)**

Time intervals (14 types) are utilized to describe at which period of time certain security properties are valid. For example, `<T1,T2>` shows that the security properties are valid exactly for the period of time from T1 to T2. The fact that the shared key between Alice and Bob is valid from 9am to 5pm can be specified as follows:

```
Shared(key,Alice,Bob),Time(<9am,5pm>).
```

Other time intervals have different starting and ending times but their structures are quite similar.

**(Time sequences)**
Time sequences are used to describe periods of time when security properties are valid. Members of a time sequence are represented by an ordered list. For example, *Alice* and *Bob* can use the shared key from 9am to 10am and from 5pm till 10pm:

\[
\text{Shared(key,Alice,Bob),Time(<9am,10am>,<5pm,10pm>).}
\]

**Probability**

The class ‘*ProbabilityFunc*’ specifies probability which is utilized if it is impossible to predict exactly what will happen with security properties of components or services, when they will demand certain services from other components or services, etc. To describe probability, we employ an interval from 0 to 1 ([0,1]). For example, the fact that the shared key between Alice and Bob is valid from 11am to 4pm with the probability from 70% to 90% can be defined as follows:

\[
\text{Shared(key,Alice,Bob),Time(<11am,4pm>),Probability([0.7,0.9]).}
\]

### 4.4.10 Relationships among security parameters

There are several parameters used in security functions that have been presented in Tables 4.1-4.6. The relationships among them are illustrated in Figures 4.25 and 4.26.

We name them here again and draw relationships among them based on information from the previous sections.

The classes that represent security parameters are subclasses of the class ‘*SecurityParam*’:

- ‘*Person*’ – represents security subjects such as a data owner or data custodian;
- ‘*Key*’ – describes symmetric, private, and public cryptographic keys;
- ‘*Size*’ – shows the length of cryptographic keys;
- ‘*Operation*’ – is responsible for security operations such as encrypt or decrypt data;
• ‘AuditData’ – expresses audited data;
• ‘Algorithm’ – is represented by the classes ‘SecurityAlgorithm’ and ‘SecurityConcept’ from SASO;
• ‘Standard’ – is represented by the classes ‘SecurityAlgorithm’ and ‘SecurityConcept’ from SASO (similar to the class ‘Algorithm’);
• ‘Object’ – describes security objects such as a database field or a file;
• ‘State’ – shows a state of transactions or security operations;
• ‘Role’ – demonstrates roles played by subjects such as a security administrator or a manager;
• ‘Website’ – specifies various websites;
• ‘Action’ – defines actions performed on certain objects by certain subjects;
• ‘Method’ – is another name of a security algorithm (‘Algorithm’) or standard (‘Standard’);
• ‘Event’ – expresses security events such as an unauthorised attempt to read or write to a password file;
• ‘Condition’ – some security conditions.

Figure 4.26. Relationships among security parameters
4.5 Relationships between concept classes across security ontologies

In this section, we summarise and illustrate relationships among different security classes included in SASO and SFO and explain briefly how various security concepts can be used for specifying security properties of component-based (service-based) software systems and their distributed constituent components or services.

Figure 4.27 depicts relationships among the different security concepts described above. The main class of SFO is the class ‘SecurityFunction’ that consists of six subclasses including ‘CryptoSupportFunc’, ‘IdentificationAuthorisationFunc’, ‘SecurityAuditResourceUtilFunc’, ‘UserDataProtectionFunc’, ‘PrivacyFunc’, and ‘TrustedChannelFunc’ introduced in the previous sections.

The class ‘SecurityFunction’ include security functions which have the property ‘hasSecParams’ that can contain the classes ‘SecurityConcept’ (from SASO), ‘SecurityAlgorithm’ (from SASO), and ‘SecurityParam’ as values. Only these three classes are used as values since they define security algorithms and standards as well as security parameters such as cryptographic keys or operations. Moreover, the classes ‘TimeFunc’ and ‘ProbabilityFunc’ can be utilised to specify security of dynamic software systems. Since many security algorithms and concepts have various assurance levels and use security credentials for access control, some of SASO classes (the classes ‘SecurityConcept’ and ‘SecurityAlgorithm’) have two properties including ‘hasSecCredential’ and ‘hasSecAssurance’ which have classes ‘SecurityCredential’ and ‘SecurityAssurance’ as values accordingly. The classes

![Figure 4.27. Relations between various classes](image-url)
‘SecurityFunction’, ‘SecurityConcept’, ‘SecurityAlgorithm’ and combinations of their subclasses assure certain complicated system’s security objectives defined by the class ‘SecurityObjective’.

Interfaces of services or components are specified by the class ‘Service Interface’ that has the property ‘hasPropValue’ which has the class ‘Prop’ as a value. The class ‘Prop’ characterises service’s or component’s functional (the class ‘FuncProp’) and non-functional (the class ‘NonFuncProp’) properties. Security properties are specified by the class ‘SecProp’ which is the subclass of the class ‘NonFuncProp’ and is used as a value for the property ‘hasSecFuncValue’ belonged to the class ‘SecurityFunction’.

As it can be seen, relationships between SASO and SFO are logical and not complicated. At the same time, dependencies among SAO, SDO, SAVO, SFO, and SASO (see Figure 4.1) are also rather simple and logical from the perspective of a software security engineer.

4.6 The use of security ontologies

Ontologies, as the representation of the domain knowledge, explicitly formalise and specify of the concepts and their corresponding relationships. Also, they include associated specific instances for the corresponding concepts. These instances contain the actual data used in knowledge based applications [PSL03]. Hence in this section, we describe how information needed for using our security ontologies can be gathered and how these ontologies can be filled with this information.

First, we should mention that security experts are needed [TG06] to fill the security ontologies with the required information and then to deploy them (e.g., security controls) in an organisation that uses our software system. Their decisions may be influenced directly, e.g., by various organisation policies, results of risk analysis, Service Level Agreements (SLAs), and infrastructure information. On the other hand, these decisions may be affected indirect ways by best practices, security standards, security consulting from vendors and independent sources, security websites [SF07], security mailing lists [SF07], and vulnerability catalogues [CVE07].

The process of using the security ontologies and filling them with the required data consists of nine stages:

1. Gathering infrastructure data regarding the network structure, software applications, launched servers, hardware, versions, active services and ports using network scanners such as Nmap [Nmap07] and other tools from Backtrack2 [Backtrack2] or Backtrack 3;

2. Selecting and evaluating assets using quantitative and qualitative risk analysis, as previously described;

3. Finding and ranking vulnerabilities using penetration tools including from Backtrack2 or Backtrack3, CORE IMPACT [CORE08] or SAINT [SAINT08];
4. Making a trade-off between security-related information gathered on the previous stages and the business needs;
5. Collecting data related to possible security attacks and possible variants of countermeasures;
6. Extracting information from security policies, evaluating them using the knowledge of the security expert, and correcting (if needed) security requirements;
7. Associating security requirements with specific security controls using introduced security functions;
8. Filling the security ontologies with the data collected previously;
9. Iterating from the first stage to the last stage on a timely basis.

It is worth mentioning that the security ontologies should be stored somewhere to allow to access them securely and easily at the same time. We discuss this issue in the next chapters.

## Summary

Users of software systems, component or service based systems in particular, require from them more features, flexibility and better protection. Hence, these software systems become highly complicated and increasingly distributed. At the same time, a new generation of security attacks, especially distributed multi-phased attacks, appear. Such attacks are quite difficult to identify and mitigate. One of the possible solutions to detect such attacks is to allow a constituent system’s components to collaborate. Components should have a common vocabulary to allow them to exchange information regarding security attacks and countermeasures. Therefore, we have utilised an ontological approach, security ontologies in particular, to specify information security issues in a way understandable to both humans and software agents.

Moreover, in this chapter we have presented the detailed description of the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), and the security function ontology (SFO) that are the first three ontologies from the series of our security ontologies. They are created using our knowledge regarding information security and software security engineering. Our security ontologies map high-level security requirements such as security policies and system security goals on low-level technical implementations, in other words they separate security requirements (“What”) from their implementations (“How”) mapped on required actions (“Do”).

In the next chapter, we introduce other security ontologies including the security attack ontology (SAO) and the security defence ontology (SDO).
Chapter 5

Security attack and defence ontologies

The previous chapter was an introduction to our security ontologies where we explained our motivations for developing and using them and reviewed general concepts and techniques that underlie their design and implementation. Besides, we described the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), and the security function ontology (SFO). In this chapter, we define the security attack ontology (SAO) and the security defence ontology (SDO) which can be used as a common vocabulary by coalitions of defensive components (DCs). Various DCs use SAO and SDO for interacting with each other and sharing a common understanding of information about attacks and approaches to resist them. In particular, coalitions can be very useful for detecting distributed multi-phased attacks such as the Mitnick attack [UJF+04]. Both SAO and SDO can evolve over time as well. For example, if a DC from a coalition detects a new attack, it adds the attack as a new class to SAO and shares the ontology with other members of the coalition. When any coalition member develops a new countermeasure against an attack, then it is added to SDO as a new class and distributed among other members. It is worth mentioning that both security ontologies have been developed based on our knowledge about security attacks and defences.

5.1 SAO and SDO classes

The main SAO class, called ‘Attack’, has five subclasses, as illustrated below in Figure 5.1: ‘WSAttack’, ‘P2PAttack’, ‘DoSAttack’, ‘SniffingAttack’, and ‘MultiPhasedDistributedAttack’. In particular, MultiPhasedDistributedAttack employs security attacks from the first four subclasses. The class ‘WSAttack’ defines attacks on Web services (WSs) while the class ‘P2PAttack’ specifies security attacks against the peer-to-peer (P2P) systems. The class ‘DoSAttack’ describes various denial of service (DoS) attacks. Security attacks on communication channels are characterized by the class ‘SniffingAttack’. An example of multi-phased distributed attacks is the Mitnick attack [UJF+04], which is the subclass of the class ‘MultiPhasedDistributedAttack’ and which can be performed through the use of attacks from the classes ‘DoSAttack’, ‘WSAttack’, ‘SniffingAttack’, and ‘P2PAttack’. Only these five classes of attacks are considered since our system might be vulnerable to them. Other types of security attacks such as physical security attacks (related to attacks against hardware resources or humans) or social engineering (psychological attacks against humans) are not taken into account.
Chapter 5 Security attack and defence ontologies

since they are beyond of the scope of this thesis. The class ‘Attack’ (and its subclasses) has a property called ‘hasRelation’ which relates to the class ‘Defence’ (and its subclasses) from SDO, as depicted in Figure 5.1. Using this property it is possible to define the rules of anti-correlation, i.e., a selection of proper defenses against a security attack. The property ‘hasDescription’ is utilised to specify the particular attacks and defences. In case of the attack definitions, this property may also contain the correlation rules, i.e., the rules that specify how to detect security attacks that are possibly related to each other and identify the triggering root attack.

**Figure 5.1. The class ‘Attack’ and ‘Defence’**

The class hierarchy of SDO is similar to SAO. The main SDO class is called ‘Defence’ and also has subclasses (Figure 5.1) that correspond to SAO classes and contain defenses against attacks on Web service (the class ‘WSDefence’) and P2P systems (‘P2PDefence’) attacks, safeguards against DoS attacks (the class ‘DoSDefence’), and countermeasures against sniffing attacks (the class ‘SniffingDefence’) and multi-phased distributed attacks (the class ‘MultiPhasedDistributedDefence’). The class ‘Defence’ has two properties ‘hasRelation’ (ties to the class ‘Attack’) and ‘hasDescription’ that utilises the security algorithm-standard and security function ontologies to specify countermeasures, as described in the previous chapter.

### 5.2 The classes WSAttack and WSDefence

In this section, which is partially based on our research paper [VH06a] and [SS05,Lin04,Neg04,Fau03], we describe the classes ‘WSAttack’ and ‘WSDefence’.

#### 5.2.1 Web service attacks and defenses

Web services (WSs) were designed to allow various technologies to be utilized at the WS layer (the application layer of the OSI model). They have been deployed recently by many companies
including governments, banks, large corporations, etc. because of their simplicity of use, platform independence, XML/SOAP support, rich functionality and interoperability. However, Web services also have raised many new unexplored security issues as well as new ways of exploiting inherited old security threats. Semantic Web services, which can publish the information about their functional and non-functional properties, add even more security-related problems. Attackers do not need to scan the Web in order to find targets because they could go to UDDI Business Registry (UBR) and get all the information they need for fulfilling Web service attacks.

Actually, the whole Web service attack tree consists of several stages during which the attacker discovers weakness, then penetrates the WS layer and further gets access to mission critical applications and infrastructures. For example, the XML injection attack [SS05] occurs when user input is passed to the XML stream without proper data verification. This attack can be stopped by scanning the XML stream. Another type of attacks against Web services is the denial of service (DoS) attack which takes place when the attacker sends extremely complicated but legal XML documents forcing the system to create huge objects in memory and deplete system’s free memory.

We specify five attack classes for the WS layer, as illustrated in Figure 5.2, including ‘DiscoveryAttack’, ‘ApplicationAttack’, ‘SOAPAttack’, ‘XMLAttack’, and ‘SemanticAttack’. We consider the class ‘ApplicationAttack’ because many application-related security threats have been “reborn” in the Web service context. ‘SemanticAttack’ describes new types of attacks that particularly target the new generation of Web services, semantic Web services. For example, the attacker can create a very complicated ontology that can crash an OWL parser. Examples of ‘DiscoveryAttack’, ‘SOAPAttack’ and ‘XMLAttack’ are UDDI attacks, SOAP replay attacks and XML injection respectively. In order to reach their malicious goals some smart attackers might create new attacks by combining security attacks from different classes.

The first step for the attacker after finding a possible target (i.e., Web services) utilising UDDI Business Registry (UBR) is to discover points of weaknesses in WSDL documents which can be used as a vulnerability guide book for getting access to mission critical applications and infrastructures. The attack stages are demonstrated in Figure 5.3.
As previously mentioned, Web services are powerful, easy-to-use and open. However, they are highly dangerous because many secrets may be exposed. Since attack methods improve continuously (more efficient DoS attacks or better automated discovery), more research work and better defence tools are required. Three layers of protection of Web services can demonstrated in the following way where the actual order is unimportant, as shown in Figure 5.4.

1. Semantic and syntax examination and validation, i.e., XML, SOAP, WSDL, OWL, OWL-S, etc documents should be examined for security threats as well as verified and validated;
2. Content inspection and policy/security properties protection, i.e., well-formed documents that pass the previous layer may still have security related issues; for example ……..
3. Protection of entities, i.e., attackers may create malicious content within documents. To prevent this, such documents as WSDL or OWL should be identified and protected in advance (e.g., using digital signatures).

### 5.2.2 The class DiscoveryAttack

The class of ‘DiscoveryAttack’ is represented by two subclasses: ‘WSProbingAttack’ and ‘UDDIAttack’, as shown in Figure 5.5. WS Probing attacks can be further subdivided into two subclasses including ‘WSDLScanningAttack’ and ‘ParameterTamperingAttack’. The class ‘WSDLScanningAttack’ define the attacks that target the WSDL interface made available by a Web service. An attacker simply scans the WSDL interface to get any sensitive information about underlying technology implementations, associated vulnerabilities, and invocation patterns. In parameter tampering attack (the class ‘ParameterTamperingAttack’), an attacker focuses on the application business logic. She tries to modify parameters used as the security measure for certain operations and left by Web service developers (hidden or fixed fields such as a parameter in a URL). Attackers can modify these parameters using various host-based proxy servers to tamper data and bypass the security mechanisms.
Since WSDL documents contain information about functions and their parameters that are available to consumers; they become an easy target for attackers. They might try to submit various parameters such as special characters in order to crash the implementation of a Web service.

Usually, UDDI and UBR are used to find Web services, however, they can also point to targets and provide all information needed to fulfil an attack on Web services, i.e., attackers do not need to scan the Internet in order to find vulnerable Web services because they just go to any UBR and find targets. Such UDDI and UBR attacks are rather dangerous because of difficulties to detect them.

5.2.3 The class XMLAttack

The class ‘XMLAttack’, as illustrated in Figure 5.6, has several subclasses including ‘ParsingAttack’, ‘XMLInjectionAttack’, and ‘XPathInjectionAttack’.

The XML injection attack (the class ‘XMLInjectionAttack’), similar to the SQL injection (the class ‘SQLInjectionAttack’) and XPath injection attacks (the class ‘XPathInjectionAttack’), occurs when user input is passed to the XML stream. Due to the fact that the XML document can be parsed by the middleware or the database (DB), XML code can be injected to the DB, and when it is retrieved from the DB, it becomes the part of the XML stream. The XML injection attack tree looks as follows, the attacker:
1. Navigates to UBR and requests for a website;
2. Attaches to UDDI and asks for a Web service and its WSDL documents;
3. Examines them and finds vulnerable methods;
4. Tests these methods in order to find possibilities for the attack;
5. Changes his/her user ID in order to crack a Web service.

Below we exemplify the XML injection attack.

```
<User>
  <ID>12345</ID>
  <Name>Bad Guy</Name>
  <Email>badguy@oops.com</Email>
  <Addr>Bad St</Addr><ID>0</ID><Addr>Bad St</Addr>
</User>
```

The result of parsing by SAX parsers is ID equals zero. In the beginning a unique user ID equals 12345, however, the attacker enters also a fake street address (Bad St</Addr><ID>0</ID><Addr>Bad St). After parsing the address, user ID is rewritten (ID=0) because SAX parsers allow overwriting earlier nodes (SAX Attacks). DOM parsers are more complicated and intelligent and can withstand the XML injection attacks, however, they cannot resist against other types of attacks including DoS attacks when hackers can send extremely complicated but legal XML documents (the class ‘DOMAttack’). It forces the system to create huge objects in memory and deplete free memory. The XPath injection attack (the class ‘XPathInjectionAttack’) is similar to the XML injection attacks and explicitly described in [SS05].

CDATA field allows to include illegal characters in XML documents. It can lead to several attacks represented by the class ‘CDATAFieldAttack’ including XML, XPath or SQL Injection attacks and Cross Site Scripting (XSS) attacks, as illustrated in Figure 5.7

![Diagram of攻撃の類型]

**Figure 5.7. The class ‘CDATAFieldAttack’**

Usually, software developers assume that certain data types cannot be embedded in XML. However, such assumptions may lead to various vulnerabilities. For example, many commercial
parsers strip the CDATA resulting in a string that contains the non-escaped dangerous characters. Below we demonstrate how the script language can be used for malicious purposes:

```xml
<TEST>
  <![CDATA[<SCRIPT><![CDATA[
  alert('u r hacked')
  ]]>]]></SCRIPT><![CDATA[>
</TEST>
```

XML parsers usually strip “<” or “>” symbols, however, the CDATA field permits to include such characters. In our case, the result of parsing of the given example is following:

```xml
<TEST><SCRIPT>alert('u r hacked')</SCRIPT></TEST>
```

And then, the user sees the message “u r hacked”. This is just a proof of concept script but with some modifications it can be used to steal cookies or spy on the user [FGH+07,SP08].

5.2.4 The class WSDoSAttack

Web service denial of service attacks are represented by the class ‘WSDoSAttack’ which composed of several subclasses, as illustrated in Figure 5.8.

![Figure 5.8. The class ‘WSDoSAttack’](image)

The main idea of coercive parsing attacks (the class ‘CoerciveParsingAttack’) is to exploit XML-based parts of Web services. The classes ‘ReplayAttack’, ‘RecursivePayloadsAttack’, and ‘OversizePayloadsAttack’ are the subclasses of the class ‘CoerciveParsingAttack’. During the replay attack, the attacker sends repetitive SOAP messages in order to overwhelm a Web service. This type of attacks is rather difficult to detect since HTTP requests are well formed and IP addresses and network packets are valid. The capability of XML of embedding the elements within documents leads to the recursive payloads attack. The attacker simply creates a document with a huge number of nested elements in-depth in order to break the XML parsing
mechanism. The overseize payloads attack occurs when the attacker creates a large XML-based document which cannot be processed by the XML parser due to the lack of free memory.

The class ‘ExternalReferenceAttack’ is another subclass of ‘WSDoSAttack’ that can be subdivided into three subclasses: ‘ExternalEntityAttack’, ‘SchemaPoisoningAttack’ and ‘RoutingDetoursAttack’.

The ability of SOAP and OWL/OWL-S documents to include external entity references allows them to organize themselves dynamically. However, attackers can create documents with references to malicious content such as external ontologies in order to fulfill external entity attacks (the class ‘ExternalEntityAttack’).

The schema poisoning attack (the class ‘SchemaPoisoningAttack’) can be described as follows: XML schemas contain formatting instructions for XML parsers when they interpret XML documents. If the attacker modifies the content of the schema he can create a WS DoS attack by manipulating data types, encoding instructions, etc.

As for the routing detours attack (the class ‘RoutingDetoursAttack’), the WS-Routing specification explains how to direct SOAP messages through series of intermediaries in complex Web service environments by using special routing tags with routing instructions. However, the attacker can capture one of the intermediaries and insert malicious routing instructions enforcing sending confidential documents to a fake location. For example, the man-in-the-middle (MITM) attack (the subclass of the class ‘RoutingDetoursAttack’) can be realized if the attacker gets confidential document, then removes malicious instructions, and resends its original version to its original destination. Also, the WS DoS attack can be launched if the attacker routs a document to a non-existent destination.

5.2.5 The class ApplicationAttack

Some of the traditional application attacks (the class ‘ApplicationAttack’) are illustrated in Figure 5.9. The others such as the buffer overflow attack can be found in [FGH+07,SP08, WikiBO07].

For example, the SQL Injection attack, which is similar to the XML or XPath Injection attacks, can be launched if the attacker executes multiple instructions for a database using SQL separators.
5.2.6 The class SOAPAttack

The class ‘SOAPAttack’ is shown in Figure 5.10. For example, the SOAP header attack (the class ‘SOAPHeaderAttack’), which is used in the WS DoS attack, can be realized if the attacker creates SOAP messages with very complex SOAP headers. Another subclass of the class of the SOAP attacks is the SOAP replay attack (the class ‘SOAPReplayAttack’) that comes about when the attacker sends repetitive SOAP messages on purpose to overburden a Web service.

In SOAP header action attack (the class ‘SOAPHeaderActionAttack’) the attacker tries to bypass protections that rely on the SOAPAction field.

Since the SOAP protocol is stateless (similar to the HTTP protocol) and transport independent, the software developers have to implement their own state mechanisms including in-line session
IDs and cookies in headers. Hence, old attacks such as prediction of the session ID (the class ‘PredictableIDAttack’) still work. Cookie attacks (the class ‘CookieAttack’) include stripping cookies at the web servers (the class ‘CookieStrippingAttack’) and improper routing of cookies (the class ‘CookieBadRoutingAttack’). These other attacks involve

5.2.7 The class SemanticAttack

The semantic attack (the class ‘SemanticAttack’), as illustrated in Figure 5.11, is a new emerging class of attacks which are based upon utilizing the structure and processing rules of SOAP, WS-* and any other semantic web standards including OWL [OWL07], OWL-S [OWLS07], SWL [SWL07], KAOs [KAOS07], METEOR-S [METS07], WSMO [WSMO06], and WSDL-S [MVR+04].

Figure 5.11. The class ‘SemanticAttack’

The semantic WS DoS attack (the class ‘SemanticWSSDoSAttack’) occurs when the attacker creates huge or very complicated files containing ontologies or schemas in order to break a parser. The semantic discovery attack (the class ‘SemanticDiscoveryAttack’) leads to information leaks due to the fact that semantic data might expose information about systems infrastructure and policies (especially security policies). Semantic crypto attacks (the class ‘SemanticCryptoAttack’) such as attacks on signatures might allow malicious SOAP bodies to be signed by valid signatures; or bogus signatures can be utilized to sign valid SOAP bodies. The new generation of attacks including the semantic WS-* attacks (the class ‘SemanticWSStarAttack’) are used for breaking Web service standards such as WS-Security, WS-Policy, WS-Trust, WS-Addressing, etc. The semantic ontology attack raises even more security issues for the future because the mass usage of semantics in the Web 2 [OR05].

5.2.8 The class WSDefence

The class ‘WSDefence’, as depicted in Figure 5.12, includes several subclasses (more details about countermeasures can be found in [SS05]).
The class ‘WSDefence’ has five major subclasses including ‘DiscoveryDefence’, ‘ApplicationDefence’, ‘SOAPDefence’, ‘XMLDefence’, and ‘SemanticDefence’ and uses the class ‘WSStandards’. The class ‘ParsingDefence’ contains classes that specify countermeasures against parsing attacks while the class ‘WSDoSDefence’ defines methods for resisting WS DoS attacks. Both classes are subclasses of the class ‘XMLDefence’. The class ‘WSStandards’ is the subclass of the class ‘SemanticDefence’ and specifies how to protect Web services using Web service standards such as WS-Security (the class ‘WSSecurity’), WS-Policy (the class ‘WSPolicy’), WS-Trust (the class ‘WSTrust’), WS-Authorisation (the class ‘WSAuthorisation’), WS-Privacy (the class ‘WSPrivacy’), WS-SecureConversation (the class ‘WSSecureConversation’), and WS-Federation (the class ‘WSFederation’).

**Figure 5.12. The class ‘WSDefence’**

All subclasses of the class ‘WSDefence’ may utilise the classes ‘WSDoSDefence’, ‘WSStandards’ and ‘ParsingDefence’ as well. For instance, countermeasures against XML parsing attacks include:

- Examining of structural validity of XML documents according to the XML specification (the class ‘ExamineStructuralValidity’);
- Analysing and validating XML Schema Definition (XSD) files that are utilized to describe the structure of XML documents (the class ‘AnalyseValidateXSD’);
- Block XML content-level attacks (restrictions to XML elements) (the class ‘BlockXMLContentLevel’).
Defenses against WSDL parsing attacks include analysing and validating WSDL files (the class 'AnalyseValidateWSDL') and finding external schemas within them (the class 'AnalyseExternalSchemas'). Other subclasses of the class 'ParsingDefence' are:

- The class 'BlockDTDRecursion' defines how to block DTD-based attacks such as recursion attacks;
- The class 'BlockHTTPGet' and its subclass 'BlockContent' specify methods for blocking attacks carried in HTTP GET parameters and malicious and unexpected content;
- The class 'BlockSOAPAttachment' specify how to block SOAP attachments attacks (attachments may contain macro-viruses etc) and is used by the class 'SOAPDefence';
- Blocking brute-force attacks are described by the class 'BlockBruteForce' and can be subdivided into blocking large XML messages (the class 'BlockLargeXMLMsg') and blocking repeated attempts (the class 'BlockRepeatedAttempts');
- The class 'OntologyDefence' explains how to protect documents containing ontologies and is utilised by the class 'SemanticDefence'. Ontologies can be protected by blocking nesting (the class 'BlockNesting') inside ontology files (for example OWL files) and forbidding large ontologies (the class 'BlockLargeFiles').

Another large subclass of the class 'ParsingDefence' is 'MessageDefence', as illustrated in Figure 5.13, which defines how massages should be protected.

![Figure 5.13. The class ‘MessageDefence’](image)

A defence for XML and SOAP messages includes several techniques:

- Response messages should be encrypted/decrypted when they are passed to the client Web service (the class ‘EncryptionDecryption’);
- Message size should be checked (the class ‘MsgSizeChecking’);
- Attachments should be analysed (the class ‘AttachmentAnalysis’);
- Schema should be validated (the class ‘SchemaValidation’);
- Content should be validated (XPath) (the class ‘ContentValidation’);
- Data should be converted properly (the class ‘DataConversion’);
- Messages should be signed in order to ensure non-repudiation (the class ‘MsgSigning’);
• Time stamping should be applied for messages (the class ‘MsgTimeStamping’).

The protection against attacks on discovery mechanisms are defined by the class ‘DiscoveryDefence’ (see Figure 5.14) that includes such subclasses as ‘BlockSearchEngines’ (search engines should not be allowed to index directories that host Web services), ‘WDSLNoPublishing’ (WDSL documents should not be published on UDDI), and ‘ACLOnUDDI’ (WSDL documents published on UDDI should be protected by access control lists (ACL) and accessed only by authorised users).

![Figure 5.14. The class ‘DiscoveryDefence’](image)

Another class that we previously briefly introduced is called ‘WSStandards’, as depicted in Figure 5.15, that consists of several subclasses including ‘WSSecurity’, ‘WSPolicy’, ‘WSTrust’, ‘WSAuthorisation’, ‘WSPrivacy’, ‘WSSecureConversation’, and ‘WSFederation’. Here, we briefly introduce these classes and Web service standards they represent, while the full description can be found in [Erl04,IM02].

![Figure 5.15. The class ‘WSStandards’](image)

WS-Security (the class ‘WSSecurity’) specifies how signature and encryption headers as well as security tokens can be attached to SOAP messages [WSS02]. WS-Policy (the class ‘WSPolicy’) describes the capabilities and constraints of the security policies on intermediaries and endpoints. WS-Trust (the class ‘WSTrust’) introduces a framework for trust models to allow Web services to establish trust relationships. WS-Privacy (the class ‘WSPrivacy’) describes several models including a model for defining how Web services and requesters state subject privacy preferences, how a privacy language may be embedded into WS-Policy, how WS-Privacy may be used with a message, and how WS-Trust mechanisms can be employed to
evaluate these privacy claims. WS-SecureConversation (the class ‘WSSecureConversation’) presents how to manage and authenticate message exchanges between parties. WS-Federation (the class ‘WSFederation’) describes how to manage and broker the trust relationships in a heterogeneous federated environment. WS-Authorisation (the class ‘WSAuthorisation’) specifies how to manage authorisation data and policies. To conclude, these Web service standards are used in our approach as building blocks for creating countermeasures against Web service attacks.

5.3 Classes P2PAttack and P2PDefence

The basics of peer-to-peer (P2P) technologies have been introduced in Chapter 3 while in this section we describe attacks against P2P systems (since our system utilises the P2P network architecture) and countermeasures to resist them. The detailed description of these attacks and countermeasure can be found in Appendix B.

5.3.1 Attacks against P2P and countermeasures

P2P networks have many advantages over the client/server paradigm such as an inexpensive way to exchange information, facilitating distributed computing, providing resilience to security threats, avoiding performance bottlenecks, and achieving high scalability. However, P2P systems suffer from inherited vulnerabilities rooted from lacking of centralized monitoring and dynamics of peer membership. They also face external attacks targeting their nodes, routing mechanisms, etc. Examples of attacks on P2P systems include man-in-the-middle attack, self replication, pseudspoofing and shilling [DVP+02].
The majority of P2P attacks (the class ‘P2PAttack’) is illustrated in Figure 5.16 while countermeasures are represented by the class ‘P2PDefence’ in Figure 5.17. We introduce and explain both classes further. As you can see, their class hierarchies are quite similar and most of the classes from SAO correlate with the SDO classes.

5.3.2 Classes P2PInfrastructureAttack & P2PInfrastructureDefence

In infrastructure attacks (the classes ‘P2PInfrastructureAttack’ and ‘P2PInfrastructureDefence’), as illustrated in Figures 5.18 and 5.19, the attacker tries to fragment the network, so that it can not function any more. The infrastructures in the network involve nodes and routing systems as well.

In the structured P2P system, the DHT overlay network over the network has already solved the problem of node failure, so we focus on the attack on nodes (the classes ‘P2PNodeAttack’ and ‘P2PNodeDefence’) in unstructured P2P systems.

If a node’s available bandwidth is drained by transferring useless messages that are created by a malicious node, all of the other resources that the node is able to offer (such as CPU and storage) will also be unavailable to the P2P network. Hence, the above-explained scenario is what is called the deplete bandwidth attack (the class ‘P2PDepleteBandwidthAttack’). This is a
typical network-layer DoS attack, hence, the firewalls should be used (the class 'P2PDepleteBandwidthDefence').

Centralized P2P systems, such as Napster, are prone to a single point of failure. However, the most widely used non-centralized P2P system, such as Gnutella, KaZaA, and structured P2P system are born resilient to the chosen-victim attack (the classes 'P2PChosenVictimAttack' and 'P2PChosenVictimDefence'), due to their flooding query mechanism or DHT lookup and backup mechanisms.

Gnutella is resilient to a single point of failure, however, attackers can selectively shut down highly connected nodes (the class 'P2PHighlyConnectedNodesAttack'). These security attacks occur because P2P networks are considered to be scale-free networks. For protecting (the class 'P2PHighlyConnectedNodesDefence') against such attacks there should be overlay exponential networks (they should have enough random nodes to create an exponential network and during attacks each lost node should be replaced with a node from the exponential network) and during attacks they should be switched on.

Unstructured P2P systems are resilient to the routing attack (the class 'P2PRoutingMechanismAttack') in nature. Therefore, we mainly consider the routing attacks on DHT routing systems, and the corresponding mechanisms to maintain resilience upon these attacks. The detailed definition of the incorrect lookup routing attacks, the incorrect routing updates attacks, the partitioning attacks and countermeasures can be found in Appendix B.

5.3.3 Classes P2PSemanticAttack and P2PSemanticDefence

In P2P semantic attacks (the class 'P2PSemanticAttack'), attackers do not aim to knock down the entire system but to try to make the network inefficient or faulty, so that the users can not use it any more and switch to another system. The P2P semantic attacks, as illustrated in Figure 5.20, include flooding attacks, storage and retrieval attacks, and Face/Off attacks. Defenses against P2P semantic attacks are represented by the class 'P2PSemanticDefence', as shown in Figure 5.21.

![Figure 5.20. The class ‘P2PSemanticAttack’](image-url)
In unstructured P2P systems, such as Gnutella, multiplicative queries are broadcasted between peers. In this so called the query flooding routing mechanism, it is likely that each query generates many sub queries that overload in the network. A malicious node may take advantage of the flooding property and conduct the flooding attack (the class ‘P2PFloodingAttack’) by sending a number of bad queries to the network. A load balancing approach (the class ‘P2PFloodingDefence’) was proposed in [DG02] to mitigate the flooding attack in Gnutella with super peers.

In storage and retrieval attacks (the class ‘P2PStorageRetrievalAttack’), intruders join and participate in the lookup protocol correctly, but misinform about storage, deny access to stored data and return incorrect data. For protection (the class ‘P2PStorageRetrievalDefence’), users should observe wrong results, DoS, etc. By avoiding single point responsibilities and verifying queries from different sources, storage and retrieval attacks can be identified and resisted. To avoid one single node from being responsible for replication or facilitating access to the replicas, each client should have the ability to independently determine correct nodes to contact for replicas.

Finally, in face/off attacks (the class ‘P2PFaceOffAttack’), the attacker has inconsistent behaviours but lies intelligently by just showing a good face to one part of the network and another face to the rest. There are mainly two mechanisms to solve the face/off problem (the class ‘P2PFaceOffDefence’):

- Public Key solution (the class ‘PublicKeySolutionDefence’): To confirm others that a peer has inconsistent behaviours, other peers should have the evidence to show (public keys can solve the problem). By requiring peers to sign all of their responses, a report that contains the inconsistent response could be verified, and hence the inconsistent behaviours can be proved.
- Byzantine Protocol (the class ‘ByzantineProtocolDefence’): The face/off problem is in fact the Byzantine General Problem [LSP82,Vel02], which meant to find out the traitors among the generals.
5.3.4 Classes P2PNetworkAttack and P2PNetworkDefence

P2P systems suffer from inherited vulnerabilities rooted from lacking of centralized monitoring and dynamics of peer membership. Also they face external attacks targeting their nodes and routing systems (the class ‘P2PNetworkAttack’) as illustrated in Figure 5.22. For example, the man-in-the-middle attack, the self replication attack, the pseudospoofing attack and the shilling attack [DVP+02] are some of the attacks on P2P systems presented here. The class ‘P2PNetworkDefence’ that define countermeasures against P2P network attacks are shown in Figure 5.23.

![Figure 5.22. The class ‘P2PNetworkAttack’](image)

![Figure 5.23. Defence against P2P network attacks](image)

For example, the man-in-the-middle (MITM) attack (the class ‘P2PMTMAttack’) is referred to the fact that a malicious peer can be in the path between two honest peers. For neutralization of such attacks more secure protocols should be used (the class ‘P2PMTMDefence’). While the self replication attack (the class ‘P2PSelfReplicationAttack’) is based on the fact that there is no ways for verifying the source and content of messages. A malicious peer can answer positively to all requests and return bad messages. One of the ways to solve this problem is system of voting (the class ‘P2PSelfReplicationDefence’). The description of other attacks including such the pseudospoofing attacks, the shilling attacks, and the mechanisms to resist such attacks are presented in Appendix B.
5.3.5 Classes P2PAnonymityAttack and P2PAnonymityDefence

Anonymity can be defined as the ability to request content on a network without revealing the identity of the requester [BOR01]. It becomes compromised when a request can be identified as originating from a particular source, or when multiple requests can be identified as coming from the same source. Sender / Initiator anonymity is provided if and only if it is not possible for the receiver of a message to identify the original sender. Receiver anonymity is achieved if and only if it is not possible to ascertain who the receiver of a particular message is. Systems can be evaluated by degrees of anonymity as: beyond suspicion, probable innocence, possible innocence and exposed [RR98].

Types of attacks on anonymity (the class ‘P2PAnonymityAttack’), as depicted in Figure 5.24, include cache-timing attacks [FS00], timing attacks [RR98] and predecessor attacks [WAL+01].

Defenses against attacks on anonymity, as shown in Figure 5.25, are expressed by the class ‘P2PAnonymityDefence’.

The rather comprehensive definition of the attacks on anonymity and the relevant countermeasures can be found in Appendix B.

5.4 DoS attacks and countermeasures

In this section, we introduce two classes ‘DoSAttack’ and ‘DoSDefence’ that represent denial of service (DoS) attacks and defenses against them. At first, we describe only DoS attacks and its subclasses such as distributed DoS (DDoS) and distributed reflective DoS (DRDoS) attacks.
Then, we present countermeasures that allow preventing, detecting, blocking, and striking back these security attacks.

5.4.1 The class DoSAttack

The DoS attack (the class ‘DoSAttack’) refers to such scenario where the attacker tries to make certain resources too busy to answer legitimate requests. The number of DoS attacks has increased greatly in recent years and new types appear rapidly. The distributed DoS (DDoS) (the class ‘DDoSAttack’) and distributed reflective DoS (DRDoS) attacks (the class ‘DRDoSAttack’) are subclasses of the class ‘DoSAttack’ and use the similar principle. The main difference is that the attacker uses several hosts which are rather difficult to identify and block. Besides, it is worthwhile to mention that the DDoS and DRDoS attacks are critical threats to the Internet. Among the numerous types of the DoS attacks, we could name Ping Flooding, SYN Flooding, Pentium "F00F" Bug, Ping o' Death, Teardrop / New Tear, the land attack, and etc. During the land attack (the class ‘LandAttack’) the attacker sends SYN packets to a victim with the spoofed IP address (the same source and destination IP address and port number as the victim). In such situation, the system starts sending a TCP/IP session opening packet to itself (dead circle), which afterwards results in a system failure or crash.

![Diagram of the class 'DoSAttack'](image)

Figure 5.26. The class ‘DoSAttack’

Too large packets of data usually need to be divided into smaller parts for proper transmitting over the network (the network’s maximum transmission unit). The majority of old kernels examined packets that were too large, but did not check packets that were too small. The attacker could send packets that were smaller than acceptable, causing systems to reboot or freeze. This was the example of the teardrop attack (the class ‘TeardropAttack’). The class
‘SpamAttack’ is a special subclass of the class ‘DoSAttack’. Generally, spam is not a security threat but a type of the DoS attack. Spam can be specified as a term that describes unwanted email, forum or instant messages which flood the systems or networks and which may contain viruses or Trojan. Additionally, spam can consume a huge amount of Internet resources including bandwidth, CPU processing time, storage space, etc. In extreme cases, spam attacks may crash the system or block the whole subnet. Some of the DoS attacks are presented in [Mir03,SL03]. The class ‘DoSAttack’ is illustrated in Figure 5.26.

5.4.2 The class DDoSAttack

During the DDoS attack the attacker compromises some hosts and deploy daemons on those hosts (the 1st stage). At a later point (the 2nd stage), the attacker sends instructions to the daemon on the compromised hosts ordering it to begin flooding a target host with various types of traffic. The huge stream of data overwhelms the victim's hosts or routers which become unable to provide services. The services under attack are the “primary victims”, while the compromised systems used to launch the attack are the “secondary victims” or so-called agents (daemons or bots). The use of secondary victims in performing the DDoS attack provides the attacker with the ability to wage a much larger and more disruptive attack that makes it more difficult to trace back the real attacker. Computer pathogens such as viruses or worms are actively used to capture “secondary victims” to perform DDoS attacks. As a rule, DDoS attacks are critical threats to the Internet and are very popular among crackers and especially “script kiddies”. The reason of such popularity bases on the fact that this attack is easy to realize and there is no guaranteed 100% defence against it.

DDoS attacks can be subdivided into the bandwidth (the class ‘BandwidthDepletionAttack’) and resource (the class ‘ResourceDepletionAttack’) depletion attacks. The bandwidth depletion attack floods the victim network with unwanted traffic that prevents legitimate traffic from reaching the primary victim. The resource depletion attack relates to the situation in which the attacker sends packets that misuse network protocol communications in order to make the victim unable to process legitimate requests for services. The full description of these attacks and their subclasses can be read in Appendix B.

5.4.3 The class DRDoSAttack

Distributive reflective denial of service (DRDoS) attacks (the class ‘DRDoSAttack’) exploit normal mechanisms used by DNS and router update protocols (the class ‘DNSRouterUpdateProtocol’). During the DRDoS attack, the attacker sends a number of update, session, and control packets to numerous key Internet servers and routers with a spoofed address of the victim system. The result is a flood of update packets, session acknowledgement responses, or error messages sent to the victim. Such attacks cannot be prevented because of...
their nature and their usage of normal functionality of the system. If the victim blocks packets from those key Internet systems, she cuts herself off from sections of the Internet.

During Smurf attacks (the class ‘SmurfAttack’), the attacker sends packets to a system which supports broadcast addressing (a network amplifier), with the source address spoofed to the victim’s IP address.

The Fraggle attack (the class ‘FraggleAttack’) is one where the attacker sends packets to a network amplifier, using UDP ECHO packets and the port that supports character generation (chargen, port 19 in Unix systems), with the return address spoofed to the victim’s echo service (echo, port 7 in Unix systems) in order to create an infinite loop. Besides, this attack is more dangerous than the Smurf attack.

### 5.4.4 The class DoSDefence

Countermeasures for defending against DoS are defined by the class ‘DoSDefence’, as depicted in Figure 5.27.

![Figure 5.27. The class ‘DoSDefence’](image)

The class ‘DoSDefence’ contains several subclasses including two major subclasses ‘DDoSDefence’ and ‘DRDoSDefence’ described below. Actually, there is no a fundamental defence against DDoS and DRDoS attacks.

The class ‘TeardropDefence’ specifies countermeasures for resisting the teardrop attack, which include mainly patching, fixing security holes and utilising firewalls.

The class ‘LandDefence’ defines the land attack which can be stopped by using firewalls, patching and fixing “bugs”. Firewalls should intercept and scan the poison packet which should be dropped.

A defence against spam is represented by the class ‘SpamDefence’ include using intrusion detection and prevention systems, e-mail filters and proxies to detect, trace back and block flood attempts.

### 5.4.5 The class DDoSDefence

As previously stated, there is no 100% defence against DDoS attacks, current solutions are partial. New DDoS attacks are continually being developed by attackers to break new employed
countermeasures. DDoS defence can be based on IDS and structural reconfigurations and adaptations.

![Figure 5.28. The class ‘DDoSDefence’](image)

DDoS defence mechanisms can be subdivided into three categories of countermeasures, as depicted above in Figure 5.28:

- Detecting and preventing the setup of DDoS attacks: detect and prevent secondary victims (the class ‘DetectPreventSecondaryVictims’), and detect and neutralise masters (the class ‘DetectNeutraliseMasters’);
- Dealing with DDoS attacks while they are in progress: detect and prevent potential attacks (the class ‘DetectPreventPotentialAttacks’), stop and mitigate effects of DDoS attacks (the class top ‘StopMitigateAttacks’), and deflect attacks (the class ‘DeflectAttacks’);
- Post-attack forensics (the class ‘PostAttackForensics’).

This part of our ontology is partially based on [SL03] where a taxonomy of DDoS attacks is presented. The detailed explanation is given in Appendix B.

### 5.4.6 The class DRDoSDefence

As previously mentioned, there is also no fundamental defence against DRDoS attacks (the class ‘DRDoSDefence’) (see Figure 5.29) because of their nature.

![Figure 5.29. The class ‘DRDoSDefence’](image)

They exploit normal functionalities of the Internet. If the victim blocks packets from the DNS servers or other key Internet systems she also cuts herself off from sections of the Internet. We propose to use the structural reconfiguration and adaptation (presented in Chapter 6) of the
system together with its constituent components (the class ‘StructuralReconfigDefence’) to withstand such types of flooding attacks.

Besides, several methods defined by the classes ‘SmurfDefence’ and ‘FraggleDefence’ for mitigating Smurf and Fraggle attacks include disabling directed broadcast in all network broad routers, dropping ICMP ECHO packets, filtering UDP and ICMP packets, blocking all outbound packets from the inner network that indicate a source address not contained within the inner network’s subnet block. The radical method includes configuring all host machines to ignore ICMP broadcasts entirely.

Countermeasures (the class ‘DNSDefence’) for resisting DNS attacks include defending the root DNS server database and updating it regularly, patching DNS servers, deploying root servers using “anycast” addresses that allow multiple hosts in different networks to look like a single server. Moreover, DNS updates that go over the Internet should be signed or sent over VPNs or other secure channels.

### 5.5 Classes SniffingAttack and SniffingDefence

The sniffing (also called snooping) attack (the class ‘SniffingAttack’) includes such attacks that allow the malicious user to get information about computer networks or their network traffic. Tools, used to perform such attacks, are called sniffers. A sniffer captures contents of packets travelling over networks and saves them into a storage place such as a file or a DB. Sniffing attacks primarily target the initial connections between parties to obtain usernames, passwords, shared keys, and any confidential information. As a rule, well-performed sniffing attacks cannot be detected and usually precede spoofing or hijack attacks. Sniffing attacks can be subdivided into passive (the class ‘PassiveSniffingAttack’) and active attacks (the class ‘ActiveSniffingAttack’), as illustrated in Figure 5.30.

![Figure 5.30. The class ‘SniffingAttacks’](image)

Passive sniffing attacks intercept network traffic and allow original traffic to arrive to its destination while active sniffing attacks from the other side, use various techniques to break defences and inject traffic into the network in order to apply passive sniffing attacks. There are
several methods for injecting traffic into the network including MAC (Media Access Control) address flooding (the class ‘MACAddressFloodingAttack’), spurious ARP (Address Resolution Protocol) traffic (the class ‘SpuriousARPTrafficAttack’), fake DNS responses (the class ‘FakeDNSResponseAttack’), and MITM (Man-In-The-Middle) attacks against cryptographic protocols (the class ‘MITMCryptoAttack’).

Countermeasures for preventing or stopping sniffing attacks, as illustrated in Figure 5.31, include improvement in active monitoring by IDS (the class ‘IDSMonitoringDefence’) for sniffing signatures including lost or delayed packets (the class ‘DetectLostDelayedPacketsDefence’) and additional routing hops (the class ‘DetectAdditionalRoutingHopsDefence’), and encrypting traffic over all network connections (the class ‘EncryptTrafficDefence’).

5.6 Classes of multi-phased distributed attacks and defenses

In this section, we describe the class of multi-phased distributed attacks (the class ‘MultiPhasedDistributedAttack’) and defenses. The main reason why they have gained our attention is the fact that it is rather hard to detect them and protect systems against them. The Mitnick attacks are the subclasses of the class ‘MultiPhasedDistributedAttack’, as illustrated in Figure 5.32. To detect and resist against such security attacks (e.g., the classes ‘MitnickAttack’ and ‘WSMitnickAttack’), different distributed software components such as defensive components (DCs) should operate as a coalition.
Besides, we describe how our security ontologies can be used and demonstrate them through the example gaming system.

### 5.6.1 The traditional Mitnick attack

The traditional Mitnick attack (or simply the Mitnick attack) [UJF+04] is related to the men-in-the-middle (MITM) attack and exploits weakness of the TCP protocol design in making a TCP connection between two hosts called the three-way handshake. For example, the three-way handshake between Host 1 (H1) and Host 2 (H2) can be demonstrated in the following way:

1. **H1** initiates a TCP connection with **H2** by sending a SYN packet to **H2**;
2. **H2** sends a SYN/ACK packet to **H1** in order to establish a TCP connection with **H1**;
3. **H1** receives a SYN/ACK packet and sends an acknowledgment on getting it by sending a SYN/ACK packet to **H2**. At this stage a connection is established and the handshake is completed.

Since the Mitnick attack is a multi-phased distributed attack, it can be detected and stopped only by a coalition of DCs distributed over the Web. This attack consists of several steps, as depicted in Figure 5.33. In this attack, the attacker (A) tries to attack Host 2 (H2) that trusts Host 1 (H1) using the TCP SYN attack (the subclass of the ‘DoSAttack’ presented previously) based upon the three-way handshake during initiating a TCP connection. This attack is expressed as follows (Figure 5.33):

1. For blocking communications between **H1** and **H2**, **A** starts the TCP SYN attack (the class ‘TCP SYN Attack’ (see Figure 5.26)) against **H1**;
2. **A** sends multiple TCP packets to **H2** in order to predict a TCP sequence number generated by **H2**;
3. **A** pretends to be **H1** by spoofing **H1**’s IP address and tries to establish a TCP session between **H1** and **H2** by sending a SYN packet to **H2** (Step 1 of the three-way handshake).

![Figure 5.33. The Mitnick attack](image-url)
4. **H2** responds to **H1** with a SYN/ACK packet (Step 2 of the three-way handshake), however, **H1** cannot send a RST packet to terminate a connection because of the TCP SYN attack from Step 1.

5. **A** cannot see a SYN/ACK packet from Step 4, however, **A** can apply a TCP sequence number from Step 2 and **H1**’s IP address and send a SYN/ACK packet with a predicted number in response to a SYN/ACK packet sent to **H1** (Step 3 of the three-way handshake).

Now, **H2** thinks that a TCP session is established with trusted **H1**. Consequently, **A** has a one-way session with **H2** and can try to hijack it in order to get remote access and launch remote commands. Further, it is necessary to mention, that **H1** might register the short TCP SYN DoS, while **H2** may detect an attempt to predict a TCP sequence number. However, if these hosts do not act as a coalition, they are not able to discover the Mitnick attacks.

### 5.7 Rules

As mentioned in the previous section, hosts should act as a coalition in order to identify multi-phased distributed attacks such as Mitnick attacks. However, the rules of correlation (i.e., an identification of security attacks that are possibly related to each other and detection of the triggering root attack) and the rules of anti-correlation (i.e., a selection of a required defence against a security attack) should be specified. These rules consist of different clauses which are described in details in the next chapter where we also illustrate how attacks and defenses can be specified. There are can be thousands of rules. We discuss few of them related to the WS Mitnick attack in the next section and the following chapters. We also demonstrate in this section that rules can be defined using different languages.

#### 5.7.1 Alert processing

Each attack may generate certain alerts (security events) with several attributes including the name of the attack, the attacking target, the source of the attack, time, and so on. More specifically,

- **Name** – a security attack has the attack name taken from the SAO;
- **Source and target** – the source of the attack and the attacking target are defined by several attributes including the source and target IP addresses, the source and target ports, the source and target host names, the service names, the processes etc;
- **Time** – expresses when an attack happens.

Security events can be represented by the class ‘SecurityEvent’ from SAVO. Each security event has a number of properties including ‘NAME’, ‘TARGET’, ‘SOURCE’, and ‘TIME’ which specify a name of an occurring security event, which IP address has been targeted and from which IP address an attack has been initiated, and when an attack has happened. The class ‘SecurityEvent’ is illustrated below.
Chapter 5 Security attack and defence ontologies

Then, security events should be processed somehow. In our approach, alert processing is based on the work of [CM02] and [Snort07] and uses Snort rules [Snort07]. We have changed the initial process consisted of three steps by modifying these steps and adding the fourth and fifth steps.

The first step is alert gathering: security attacks are detected by IDSs which emit alerts. These alerts are converted to records and then saved to databases.

The second step is alert clustering: alerts detected by different IDS are grouped into clusters, i.e., alerts from one cluster correspond to a particular attack. The process of finding similarities can be illustrated as follows:

- Name – two attacks are similar if their attack names are identical;
- Source and target – two sources or targets are similar if their related information is similar;
- Time – two attributes are similar if their difference is within predefined threshold.

For example, there are two events SecEv1 and SecEv2. The process of alert clustering can be demonstrated in the following way:

\[
\begin{align*}
&\text{IF}(\text{SecEv1.NAME} == \text{SecEv2.NAME}) \{ \text{SecEv1} == \text{SecEv2}; \} \\
&\text{IF}(\text{SecEv1.SOURCE} == \text{SecEv2.SOURCE} \text{ AND } \text{SecEv1.TARGET} == \text{SecEv2.TARGET}) \{ \\
&\quad \text{SecEv1} == \text{SecEv2}; \} \\
&\text{IF}(\text{ABSOLUTEVALUE}(\text{SecEv1.TIME} - \text{SecEv2.TIME}) < \text{THRESHOLD}) \{ \text{SecEv1} == \text{SecEv2}; \}
\end{align*}
\]

The third step is alert merging (correlating). On this step alerts from each cluster are merged in order to generate a global alert with merged data. This process can be described as follows:

- Classification – a union of all classification values in the particular cluster is created in order to merge attack classification for one cluster;
- Source and target – a unique (common) pair of source and target for the global alert is created if two or more pairs of sources and targets are similar. In case if two or more pairs of sources and targets are different, all these different values are added to the global alert;
- Time – time is generated based on timestamps, i.e., when an attack was detected;

This process can be demonstrated through the use of chronicles [MD03] which provide a mechanism to model event temporal patterns and monitor the system’s evolution. Besides, the same process can be expressed in SWRL [SWRL06] or in logic programming (presented in the next chapter). For example, three security events including ‘PortScanAttack’,

\[
\begin{align*}
&\text{Class SAVO.SecurityEvent} \{ \\
&\quad \text{Property NAME}; \\
&\quad \text{Property TARGET}; \\
&\quad \text{Property SOURCE}; \\
&\quad \text{Property TIME}; \}
\end{align*}
\]
‘HTTPServerOverflowAttack’, and ‘RemoteAccessAttack’ have happened one after another. This rule can be represented in the following way:

\[
\text{HTTPServerBreakAttack}(\text{?SOURCE}, \text{?TARGET}, \text{?t}_3) \iff \\
\text{PortScanAttack}(\text{?SOURCE}, \text{?TARGET}, \text{?t}_1), \\
\text{HTTPServerOverflowAttack}(\text{?SOURCE}, \text{?TARGET}, \text{?t}_2), \\
\text{RemoteAccessAttack}(\text{?SOURCE}, \text{?TARGET}, \text{?t}_3), \text{?t}_1 < \text{?t}_2 < \text{?t}_3.
\]

In this rule, three attacks have occurred such as ‘PortScanAttack’, ‘HTTPServerOverflowAttack’, and ‘RemoteAccessAttack’, where their corresponding timestamps should be in increasing order (i.e., \(t_1 < t_2 < t_3\)), and their corresponding domain attributes (i.e., source and target IP addresses) should be equal. If all these patterns as well as constraints are satisfied, then this rule is recognised and an alert is generated stating that ‘HTTPServerBreakAttack’ is occurred.

The Mitnick attack described in the previous section can be represented a similar way:

\[
\text{MitnickAttack}(\text{?SOURCE}_A, \text{?TARGET}_{H2}, \text{?t}_4) \iff \\
\text{TCPSYNAttack}(\text{?SOURCE}_{A,H2}, \text{?TARGET}_{H1}, \text{?t}_1), \\
\text{TCPSequenceNumberPredictionAttack}(\text{?SOURCE}_A, \text{?TARGET}_{H2}, \text{?t}_2), \\
\text{IPSpoofingAttack}(\text{?SOURCE}_A, \text{?TARGET}_{H2}, \text{?t}_3), \\
\text{RemoteAccessAttack}(\text{?SOURCE}_A, \text{?TARGET}_{H2}, \text{?t}_4), \text{?t}_1 < \text{?t}_2 < \text{?t}_3 < \text{?t}_4.
\]

The fourth step is the process of anti-correlation when a system represented by a human or software agent selects a defence strategy from the pool of predefined defenses, i.e., from SDO or specifies a new one. For example, the case of selecting of a countermeasure against a Mitnick attack can be represented in the following way:

\[
\text{IF(SecEv1.NAME == MitnickAttack) \{ ENFORCE MitnickDefence; \}}
\]

The same can be expressed in logic programming, as illustrated below:

\[
\text{MitnickDefence} \iff \text{MitnickAttack}.
\]

On the final step the actual countermeasures against Mitnick attacks are deployed. They can be specified as a class containing an array of actions:

\[
\text{MitnickDefence MitnickDefence1 = new MitnickDefence (SecurityEvent SecEv1) \{ 
IF(SecEv1.NAME == MitnickAttack) \{ 
\text{ACTIONS} \{ 
CLOSE_CONNECTIONS_IP=SecEv1.SOURCE; BLOCK_IP = SecEv1.SOURCE; 
INCREASE_CONNECTION_QUEUE = +50; DECREASE_TIMEOUT = -1ms; \} \} \}
\]

121
This rule states that if a Mitnick attack is identified then several countermeasures should be enforced including closing all connections from the attacker’s IP address and blocking this address, increasing the size of the connection queue and decreasing the time-out waiting for the three-way handshake.

In the next chapter, we discuss how the process described above can be applied. Besides in the next section, we demonstrate the use of this process utilising the WS Mitnick attack.

5.8 Example: WS Mitnick attack

In this section, we present the modified Mitnick attack called the WS Mitnick attack (the class ‘WSMitnickAttack’) and explain this attack through the use of the example. Also as previously mentioned, a software system’s constituent components may be highly distributed and vulnerable to various security attacks, especially distributed attacks such as Mitnick attacks which are subclasses of multi-phased distributed attacks, as shown in Figure 5.32.

For example, there is a component-based gaming system, illustrated in Figure 5.34, which consists of game servers (peers) that host software components and that need to communicate securely with each other over the network in order to allow game players to play online games and also store information about them including their credit card details and home addresses.

Such information is required because a game provider needs to be sure that game players are real people who satisfy certain criteria (for example, age). In our case, game servers are represented by Host 1 (H1), Host 2 (H2), Host 3 (H3), and Host 4 (H4) while game players are Client 1, Client 2, and Client 3. Initially, H1, H2, H3, and H4 have the same configuration by default, however, they can be customized later to support users from different regions in the world. H1 and H2 deliver services to game players from the U.S.A. while H3 and H4 support

![Figure 5.34. Mitnick attacks against a gaming system](image-url)
users from Australia. And, even if they are responsible for different geographical regions, they still need to synchronize game data for delivering reliability. Moreover, H1, H2, H3, and H4 host Web services (WS1, WS2, WS3, WS4) to allow game players to play games through the use of Web service interfaces and databases such DB1 (hosted by H1) and DB2 (hosted by H2) which store information about game players from the U.S.A. and DB3 (hosted by H3) and DB4 (hosted by H4) which contain data about users from Australia. Also, there is the manager who controls the whole system.

An attacker (A) wants to get credit card details of all game players from the U.S.A. (Client 1 and Client 2). After registering as a game player from the USA using stolen credentials and social engineering skills (beyond the scope of this work), and after analysing network traffic, A figures out that H1 and H2 are responsible for this region. One of the ways for A to get credit card details is to perform the Mitnick attack against both H1 and H2. The rule presenting this particular attack has been introduced in the previous section. It has stated that the Mitnick attack occurs if a TCP SYN attack (the class ‘TCPSYNAttack’), a TCP sequence number prediction attack (the class ‘TCPSequenceNumberPredictionAttack’), an IP spoofing attack (the class ‘IPSpoofingAttack’), and a remote access attack (‘RemoteAccessAttack’) happen one after another. It is necessary to mention, that the classical Mitnick attack can be performed as the first stage of game server penetration. The next stage is to perform the XML injection attack against WS1 that occurs when user input is passed to the XML stream without proper data verification. Also, A does not need to scan the Web in order to find Web service targets. A just goes to UDDI Business Registry and gets all required information for performing an attack against Web services such as discovering weakness in WSDL documents in order to get access to mission critical applications and infrastructures (i.e., discovery attacks represented by the class ‘DiscoveryAttack’). As previously mentioned, since the Mitnick attack is related to the man-in-the-middle attack that exploits weakness of the design of a TCP protocol in making a TCP connection (three-way handshake) between hosts. We note that a combination of security attacks against different layers such as application (Web service and P2P), network (DoS), and communication (Sniffing) layers is rather dangerous because of difficulties to detect and mitigate them.

In our case, the WS Mitnick attack consists of several stages when A tries to attack H2 that trusts H1 using the TCP SYN attack that initiates many partial TCP connections (i.e., the class ‘TCPSYNAttack’), and then penetrates the application layer using attacks against WS1 or DB1 including XML injection and WS probing attacks. The attack tree is illustrated in Figure 5.34 and is structured as follows:

1-2. A navigates to UBR and requests for a website, attaches to UDDI and requests WSDL files;
3. To block connections between \( H1 \) and \( H2 \), \( A \) pretends to be \( H2 \) (by spoofing \( H2 \)’s IP address) and starts the TCP SYN attack against \( H1 \) by opening many partial TCP connections;

4. \( A \) sends multiple TCP packets to \( H2 \) for predicting a TCP sequence number generated by \( H2 \);

5. \( A \) pretends to be \( H1 \) and sends \( H2 \) a SYN packet trying to establish a TCP session between \( H1 \) and \( H2 \);

6. \( H2 \) replies to \( H1 \) by sending a SYN/ACK packet which is not seen by \( A \), however, \( H1 \) cannot respond because of many partially opened connections caused by the TCP SYN attack;

7. Pretending to be \( H1 \), \( A \) can send a SYN/ACK packet to \( H2 \) with a predicted TCP sequence number (from Step 4) and \( H1 \)’s IP address;

8-10. (Now, \( H2 \) thinks that a TCP session is established with a trusted \( H1 \).) \( A \) can perform an attack against WS2 hosted by \( H2 \), by inspecting WS2’s WSDL files and testing vulnerable methods with XML Injection attack; if successful, \( A \) applies the XML injection to change \( A \)’s ID and get privileges;

11. If the XML injection attack is not successful, \( A \) may try an SQL injection attack against DB2 or another WS attack, because \( H2 \) still believes that it is connected to \( H1 \).

Finally, the attacker can get remote access and try to execute some commands. As it can be seen, there are some issues in the example described above. For example, if \( H1 \) and \( H2 \) do not cooperate and do not constantly exchange data about attacks, then at the initial stage of the attack, \( H1 \) registers just a short TCP SYN attack while \( H2 \) identifies only attempts to predict a TCP sequence number. Also, \( H2 \) does not know that it has been penetrated through the application layer via XML injection and WS probing attacks. So, several questions are raised:

- How can such multi-phased distributed attacks be detected?
- How should components collaborate with each other in order to resist and mitigate such attacks?
- How can countermeasures be devised?
- How can information about attacks and defenses be shared and distributed among components?

In the next section, we answer these raised questions.

5.8.1 Defence against multi-phased distributed attacks

In this section, we explain how combinations of methods described above should be used to prevent, detect and resist against multi-phased distributed attacks. We demonstrate them through the use of the gaming system’s constituent components, described in the previous
section, collaborate, identify distributed attacks, and securely share information about them and developed countermeasures using our security ontologies. The more technical description of the WS Mitnick attack is presented below.

**Scenario and analysis**

A game player ([Client 1](#)) from the U.S.A. wants to play for the first time a game called Heroes of Might and Magic 6 (HMM6). He downloads a game client application from a website [www.hmm6.com](http://www.hmm6.com). After installing online game software and specifying all requirements, she starts playing. In addition to playing online games, game players can do other activities such as chatting or sharing files. As illustrated in Figure 5.34, [Client 1](#) connects to [H2](#) and plays HMM6. At the same time, [H2](#) actively communicates with [H1](#) synchronising data. However, [A](#) tries to perform the Mitnick attack in conjunction with WS attacks, as described in the previous section, against [H1](#) and [H2](#). As mentioned above, Mitnick attacks can be detected by several members of a coalition who are distributed over the network and who cooperate with each other, therefore, [H1](#) and [H2](#) cooperate and constantly exchange data about security attacks and defenses. The rules which describe the Mitnick attack and countermeasures to resist them have been introduced in the previous section.

Assume, [A](#) attacks [H1](#) (an IP address equals 147.202.46.43) and [H2](#) (an IP address is 147.202.46.40) from a host with an IP address 192.2.1.23. All stages of the attack have been explained in the previous section. Here, we only describe how [H1](#) and [H2](#) respond to the attack. At the first stage of the Mitnick attack, [A](#) spoofs [H2](#)’s IP address and initiates TCP connections with [H1](#). Then, [H1](#) detects that many partial TCP connections are opened from [H2](#)’s IP address 147.202.46.40 at 14.40.00 on 08.08.2008. Then, one minute later [H2](#) detects that someone tries to predict a sequence number from the IP address 192.2.1.23 and spoof [H1](#)’s IP address in order to perform a Mitnick attack. To prevent this, [H1](#) sends a message (the class ‘SecurityEvent’ from SAVO) to [H2](#) to verify if [H2](#) really has opened all these connections, and if there is no response from [H2](#), then [H1](#) and [H2](#) take prompt actions, e.g., sending an alert message regarding a possible attack to other hosts (and the manager who governs the system). Properties ‘NAME’, ‘TIME’, ‘TARGET’, and ‘SOURCE’ specify a name of a security attack, when a security event has occurred, which IP address has been requested, and from which IP address the event has been initiated.

The instance of the class ‘SecurityEvent’ called ‘SecEv1’ is sent to [H2](#) and, if [H2](#) does not reply in time (e.g., after three minutes), [H1](#) generates an alert. Simultaneously, [H2](#) detects that someone from the IP address 190.2.1.23 tries to predict a TCP sequence number. Since [H2](#) receives ‘SecEv1’ and replies to [H1](#) but in turn [H2](#) does not get any acknowledgement from [H1](#), the manager concludes that a Mitnick attack is occurring and sends security events containing
this information to other hosts. Actually, there are many rules that specify attacks (similar to expert-based systems) stored in the Ontology Repository (presented in Chapter 8). In our case, the manager concludes that there is the Mitnick attack when $H_1$ detects many partial connections from $H_2$ while $H_2$ registers many TCP sequence number prediction and IP spoofing attempts following one after another. Also, $H_2$ identifies XML injection and WS probing attacks minutes later from the A’s IP address. Then, the manager concludes that all these attacks are related to each other and performed by one attacker. Consequently, the manager labels the combination of these attacks as the WS Mitnick attack, specifies the rule of this attack, and encrypts (e.g., using the AES algorithm) and sends this information to other hosts including $H_1$, $H_2$, $H_3$, and $H_4$.

It is very important to treat these attacks as a part of coordinated multi-phased distributed attacks and not as independent attacks, i.e., they should be analysed (e.g., by the manager) as a part of one complex attack in order to see, control, and maintain the whole picture of attacks. Such approach helps to identify complex attacks and develop more comprehensive countermeasures faster.

After the attack has been identified, a group of the game company’s software developers and system administrators or intelligent agents start developing countermeasures and after finding them, they securely distribute the rules of anti-correlation for this specific attack among other hosts which belong to the gaming system including $H_1$, $H_2$, $H_3$, and $H_4$. We do not explain here how countermeasures are developed and deployed. If a WS Mitnick attack is detected, then the manager enforces the system to close all connections from the spoofed IP address (147.202.46.40), then increases the size of the connection queue, decreases the time-out waiting for the three-way handshake, and blocks all connections and ports for the A’s IP address (192.2.1.23). The similar rule for describing countermeasures against TCP SYN attacks has been introduced in the previous section. It is worth mentioning that Mitnick attacks should be treated as a baseline to verify if the system can really detect coordinated multi-phased distributed attacks. In the next chapter we describe how our security ontologies can be filled with the information about various security events.

5.9 Summary

Currently, users of software applications and component-based software systems in particular, require more features, flexibility and better protection from them. Hence, they become highly complicated and increasingly distributed. At the same time, new attacks, especially distributed multi-phased attacks, which are quite difficult to identify and mitigate, appear.

In this chapter, we have demonstrated that only through the collaboration of a system’s constituent components it is possible to detect and resist against such attacks. To do so, components should have a common vocabulary to allow them to exchange information with
each other about security attacks and defenses. Therefore, we have utilised an ontological approach, security ontologies in particular, to specify information security issues in a way understandable to both humans and software agents. In this chapter, we have introduced the security attack ontology (SAO) and the security defence ontology (SDO). SAO and SDO capture relationships among security attacks and defenses against them and closely correlate with each other. Both of them have been utilised with other security ontologies with the purpose to provide a common vocabulary for various components in our system. For example, to detect any of the multi-phased distributed attacks such as Mitnick attacks, defensive components (DCs) should operate as a coalition. As long as the attack is evolving, SAO is also evolving and shared among other members of the coalition. When a new countermeasure is created by any coalition member, it is added directly as a new class to SDO which is then securely distributed among other members of the coalition.
Chapter 6

GIZKA: a language for managing security

Since our key objective is to develop an architectural approach to achieving higher level security in component (service) based software systems, we have developed the reference architecture SECROBAT with defensive components and the hybrid P2P network architecture, as described in Chapter 3. Because of the distributed nature of SECROBAT, its components have to communicate to each other regarding security attacks or defenses, or other environment changes. Hence, these software components should have a common vocabulary to share this information and utilise it for correlating multi-phased distributed security attacks and for selecting proper defenses. For this purpose, we have employed an ontological approach and created security ontologies presented in details in Chapter 4 and Chapter 5. However, to reduce the cost of developing and managing component (service) based software systems we may need to create a framework that combines various parts of our approach. A good framework lets users reuse both design and code, should be simple enough to be learned, and provide enough features. Besides, to manage SECROBAT’s security we need a formal language used as glue to link various parts of our approach. Moreover, this language should make the development and management process as secure as possible and simple and flexible at the same time. Therefore in this chapter, we introduce our language called GIZKA that allows to specify dynamic component (service) based software systems and their security properties.

GIZKA describes composition of components and allows to govern relationships and interactions between them through the use of security contracts created and verified at design time and at runtime.

GIZKA consists of three subsets described in details in the next sections including GIZKA.ESCL, GIZKA.ATTACK and GIZKA.ADL. GIZKA.ESCL lets software components specify their security properties and security contracts used to govern connections among components. For example, if a component wants to employ another component both of them should satisfy certain security requirements and capabilities of each other defined by security properties, otherwise this interaction is forbidden. GIZKA.ATTACK allows to define security attacks in a formal way and to identify, analyse, correlate, and anti-correlate them. As an example of a countermeasure against distributed security attacks, we propose to use structural reconfiguration and adaptation of the system. GIZKA.ADL is capable to specify and analyse
dynamic system (re)configurations, describe security mechanisms, security patterns, and reasoning rules for components and services. All these features allow to define programmed and runtime reconfigurations for SECROBAT-based systems to allow us to manage their security. The detailed description of GIZKA language and its features is given in the following sections.

### 6.1 Introduction to GIZKA

Currently, software components can be combined in complex composites in order to form component-based software systems (CBSSs) aimed to deliver maximum benefits in software reusability and quality (e.g., security or trust). Such CBSSs are usually composed of a number of stand-alone software components developed by third-party software vendors. Comprehensive knowledge of such components is essential to ensure that the resultant systems meet their requirements. Security has become a critical aspect of such knowledge, especially in distributed software systems. At present, software integrators have to assemble CBSSs with limited knowledge about the security characteristics of their components. For example, a general claim that a component or system is secure does not mean much and can not be relied upon. In general, only by knowing the specific security properties of the individual component can we be sure that the overall system satisfies its security requirements. This issue was addressed in [KH02,KH03,Kha05] where a security characterisation language (SCL) for specifying the security properties of software components and systems was developed.

SCL assumes that the security provisions of software components do not change over their lifetime, and consequently describes their security properties in a static manner. In dynamic adaptive software systems, the components’ behaviour and the system’s structure and behaviour can change, responding to user requirements and environment perturbations. This includes the security properties of the components and system. In particular, the specific security properties of a component or system may depend on time, have a particular probability, or even change in nature. Hence, we extend SCL (and call it as ESCL or GIZKA.ESCL) with the capability of specifying the dynamic security characteristics by incorporating such features as time, time intervals, time sequence, probability, runtime conditions, and alternative security properties. More specifically,

- Security properties with time, time intervals and time sequences state the period(s) of time when specific properties are valid;  
- Security properties with probability describe in a flexible way the security characteristics of components, where it cannot be predicted exactly which set of security properties for which components would be valid in future. Hence, security properties can be defined using the probabilistic approach;
• Alternative security properties allow the specification of components with different sets of security properties at different situations;
• Runtime conditions allow expressing security properties in a dynamic way through the employment of IF-ELSE as demonstrated in the next sections.

GIZKA.ESCL is utilised to specify and verify security contracts among software components at the design time as well as at runtime.

Another subset of GIZKA called GIZKA.ATTACK allows to specify security attacks and defenses in a formal manner in order to allow to identify attacks, analyse and correlate them to figure out the triggering root attack and the attacker’s goals, and then anti-correlate, i.e., select and deploy a proper countermeasure. Any attack has a unique signature that can be defined in GIZKA.ATTACK in a way comprehensible to software agents or humans (e.g., system administrators). GIZKA.ATTACK is closely related and based on the security ontologies such as SAO. More specifically, the process of correlation and anti-correlation consists of three stages:
• The detection of independent security attacks that are probably related to each other;
• The analysis of the root attack that may trigger independent security attacks and the prediction of the attacker’s goal;
• The creation and deployment of required countermeasures such as system’s reconfigurations (anti-correlation).

The detailed description of GIZKA.ATTACK is presented in the next sections.

Currently, the Architecture Description Languages (ADLs) such as ACME [GMW00] and Armani [Mon98] do not support the description of dynamic structural reconfigurations and adaptations of the system as well as non-fictional properties (e.g., security). Hence, we extend ACME/Armani and call this extended version GIZKA.ADL which is another subset of GIZKA and which is capable to describe the structural reconfiguration and adaptation employed as countermeasures against various security attacks. While security related features of components are expressed in GIZKA.ESCL the structure of the whole system is specified in GIZKA.ADL for depicting dynamic changes of the architecture when the security environment changes or the system’s security requirements change. More precisely,
• Clause conditions (time and event dependant) specify when and how programmed or runtime reconfigurations should take place together with a specification of what should be changed. Programmed reconfigurations capture changes that can be known at design time while runtime reconfigurations can handle the runtime changes in the system’s environment that are not defined at design time.
• Internal actions (extensions to ACME/Armani) are used to describe connections between elements including roles, ports, connectors, and functional and security properties.
- External actions (extensions to ACME/Armani) define creating, destroying and failing of elements such as components or connectors.
- Relations express runtime dependencies between architectural elements. They are used to avoid mismatches between programmed and runtime reconfigurations (the exact mechanism is described in the following sections).
- Dynamic attachments can be specified for a type or for an instance. The desired instance (e.g., a component’s instance) is selected at runtime according to certain policies.
- Additional features include security patterns, security properties, and a security baseline used for specifying dynamic reconfigurations and reasoning rules.

The complete description of GIZKA.ADL is provided in the next sections.

### 6.2 Specification of security properties

To govern connections among constituent components of component-based (or service-based) systems in order to make them secure and attack protected and specify them in a formal way, the system need various access control mechanisms. Hence, in this section we introduce GIZKA.ESCL (based on a security characterisation language capable to define static security properties) that is a subset of GIZKA. GIZKA.ESCL is used to specify and analyse security properties of dynamic software systems and their constituent components. Besides, it is capable of defining security contracts that govern the interactions among system’s components. In other words if security properties of a component-requestor (focal) and a component-responder (candidate) satisfy each other then a connection between these components is established. Also, we describe in this section how these security properties can be analysed and reasoned and present the architecture of the reasoning engine.

#### 6.2.1 Motivation example

The term “dynamic security” can be used to describe the environments which morph dynamically, and consequently, their security properties also change. The few questions regarding dynamic security can be asked:

**Why do users need dynamic security?**
- Static security does not withstand well with rapidly changing security environments (security attacks on the system, threats, policies, mission goals and different physical parameters).
- Old and current systems cannot predict and handle all future attacks and threats by their own built-in mechanisms.

**When do they need it?**
Dynamic security can be used in highly mobile and dynamic environments such as the mobile virtual office system. For example, such a system is implemented using such technologies as the peer-to-peer (P2P) technology proved to be secure and robust. Besides, there are attacks (e.g., flooding Distributed Denial of Service (DDoS) and sniffing attacks) against a peer or component that is responsible for message storage (instant messages, chat messages, audio/video messages, and emails). For resisting the sniffing attack, the system needs to encrypt these messages using shared keys distributed by key distribution peers. To prevent and oppose flooding DDoS attacks, the system uses intrusion detection (ID) peers, firewall peers, antivirus peers, and honeypots and changes its structure dynamically to improve security. It is necessary to mention, that in this example a peer is a host that hosts components that provide certain functionalities.

Furthermore, the various components of the system should have compatible security provisions to ensure that the communications between them are carried out without compromising their functionality. This requires the explicit specification of their security properties relative to their functionalities that can be expressed in GIZKA.ESCL.

For the secure exchange of emails, DES or 3DES and PGP cryptographic algorithms can be used; furthermore, for distributing cryptographic keys IKE and Diffie-Hellman, or DES and IPSec algorithms [Cry07,RSA07] can be applied. Users can switch from one algorithm to another one if network bandwidth becomes low. We focus more on dynamic changes of the system structure and dynamic changes of system security. If ID and firewall peers cannot resist DDoS attacks against Message Storage 1 (MS1) then this peer simply dies and client peers (Client 1 (C1) and Client 2 (C2)) connect to its clone (Message Storage 2), as illustrated in Figures 6.1-6.2.

![Figure 6.1. The system before attacks](image1)

![Figure 6.2. The system after attacks](image2)
SCL has provided a way for us to specify the security properties of individual components and to reason about the security compatibility between interacting components, leading to inter-component compositional security contracts. In a dynamic adaptive system context, however, further challenges remain. Firstly, SCL expresses only security properties of static systems without considering the dynamic changing nature of adaptive systems. Hence, there is a need for the language to express security properties of dynamic component-based software systems (CBSSs). Secondly, there are cases where it is not possible to predict exactly what will happen with CBSSs and when certain services will be available in future. For such purposes, probabilistic approximations can be used. They help to illustrate that certain services can be available with a certain probability at certain periods of time. Thirdly, a component may use different sets of security techniques for different situations. Consequently, the component may have different sets of security properties specified, and a focal component can select which set of security techniques (and security properties) to employ.

For example, Client1 wants to communicate securely with Client2 and shared keys between C1 and C2 and their security properties should be updated regularly (see Figure 6.1) or at certain periods of time because it reduces the chances for the system to be “hacked” even if keys are intercepted. It is also known that the network is overloaded from 9am to 5pm every day with probability of 90% then the cryptographic algorithm such as DES is used at that time for generating cryptographic keys for better performance instead of Triple-DES which is more secure but slower and which is used at other time. Besides, there can be two key distribution components which provide shared keys. However, one of them delivers this service with probability of 80% every day while another one with probability of 90% from 9am to 5pm every day and clients may select a proper component provider based on their own requirements and capabilities. Furthermore, GIZKA.ADL depicts dynamic changes of the architecture when security environment changes or a user wants to change security requirements where security properties of components are expressed in GIZKA.ESCL.

6.2.2 Introduction to SCL

As mentioned above, a CBSS may consist of several stand-alone third-party software components, which are usually provided in binary form. As such, software integrators do not usually know about their detailed security characteristics, and have to build systems with the potential of high vulnerabilities. To address this issue, the security properties of the components need to be made available so that the software integrator can assess their suitability and interoperability regarding security in the system. The security characterisation language (SCL) was introduced to satisfy this need [KH02, KH03, Kha05]. Based upon the Common Criteria for Information Technology Security Evaluation [CC06], BAN logic [BAN90, Wes01], and
extended logic programming [DBM01, DEG+01, Rob92], SCL is executable, and therefore allows automatic compatibility checking of security properties between interacting components. Furthermore, SCL allows the component developers to specify the security properties of the component in the form of logic program clauses in order to automatically check the compatibility of interacting components regarding their security properties by utilizing smodels [NS00].

In a CBSS, a software component may require that certain security requirements are met by neighbouring components and provide a set of security guarantees to these neighbouring components. We call these security requirements and guarantees ‘required’ and ‘ensured’ security properties of the component in question. Using SCL, one can specify in a logic program style the required and ensured security properties of a component as part of its interface description. In general, the required and ensured properties are parts of the tails and heads of the logic program clauses respectively.

When two components interact in a CBSS, the relevant security properties of these components need to form a ‘compatible’ compositional security contract (CsC). That is, their ensured security properties satisfy each others’ required security properties. The component that needs services provided by the other component is called the ‘focal’ component. The component that provides requested services to the focal component is called the ‘candidate’ component. The exact description of SCL may be found in [Kha05].

In the case of the mobile virtual office system described above, the component KD provides a service GetSharedKey(K, ID1, ID2), where ID1 and ID2 identify the two components sharing the symmetric key K. C1 makes a request to a key distribution component (KD) for a shared key to be used with C2: GetSharedKey(K, C1, C2). We assume that pairs of private-public keys have been already successfully distributed among participating components.

As a result, K is securely distributed between C1 and C2 to allow them to confidentially communicate to each other. The required and ensured security properties of C1 for the specified functionality are expressed as follows:

\[
\begin{align*}
\text{KeyExchanged}(K[C1, C2], KD, C1) & \leftarrow \text{KeyDistribution}(K[C1, C2], IKE, \text{DiffieHellman}). \\
\text{BelievesShared}(C1, C2, K[C1, C2]) & \leftarrow \text{Shared}(K[C1, C2], C1, C2), \text{KeyExchanged}(K[C1, C2], KD, C1).
\end{align*}
\]

The listed ensured properties of C1 suggest that the key is securely exchanged between KD and C1 if the key is distributed between C1 and KD using IKE and Diffie-Hellman algorithms. In the second rule, C1 believes the key is shared between C1 and C2 if K is shared between C1 and C2 and exchanged securely between KD and C1. Security properties involved with the functionality GetSharedKey provided by KD are:
KeyAlgorithm(K[C1,C2],DES,PGP).
KeySize(K[C1,C2],56).
KeyGenerated(K[C1,C2])←KeyAlgorithm(K[C1,C2],DES,PGP),KeySize(K[C1,C2],56).
Shared(K[C1,C2],C1,C2)←KeyGenerated(K[C1,C2]).
KeyDistribution(K[C1,C2],IKE,DiffieHellman).

The function \textit{KeyAlgorithm(K[C1,C2],DES,PGP)} specifies that a shared key K is generated with DES and PGP algorithms. It is ensured with the function \textit{KeySize} and \textit{KeyGenerated} that the shared key is generated for C1 and C2 if the algorithm DES is used, and the key size equals 56 bits. The component KD ensures these required properties by itself. The property \textit{Shared(K[C1,C2],C1,C2)} is ensured if the required property \textit{KeyGenerated(K[C1,C2])} is satisfied. Security properties of C2 for the functionality \textit{GetSharedKey(K,C2,C1)} are similar to the properties of C1:

KeyExchanged(K[C1,C2],KD,C2)←KeyDistribution(K[C1,C2],IKE,DiffieHellman).
BelievesShared(C2,C1,K[C1,C2])←Shared(K[C1,C2],C2,C1), KeyExchanged(K[C1,C2],C2,C1).

The security properties between C1 and KD regarding \textit{GetSharedKey} are compatible, and a compositional security contract between KD and C1 is formed on the following basis:

CsC(C1,KD)←BelievesShared (C1,C2,K[C1,C2]).

Similarly, a CsC between C2 and KD can be formed as:

CsC(C2,KD)←BelievesShared(C2,C1,K[C1,C2]).

It is worth mentioning that in the described example we demonstrate the use of SCL using security functionality (i.e., \textit{GetSharedKey}) which should not confuse the reader. Other functionalities such as chatting or executing transactions also can be utilised as presented in the next sections.

\textbf{Issues}

As previously mentioned, SCL does not take into account the dynamic nature of CBSSs, and consequently specifies their security properties in a static way. In general, CBSSs can be highly dynamic and adaptive with the changeable system’s structure in response to users’ behaviour or environment changes, therefore, there is the need to enhance SCL with additional constructs to define CBSSs and their dynamic security properties. We introduce our work in the following sections.
6.2.3 Extended SCL (GIZKA.ESCL)

As previously mentioned, SCL assumes that the security properties of a CBSS and its software components do not change which is not true. As a rule, the system’s and components’ structure and behaviour can change over time due to environment change or requirement changes. Static security properties are not adequate for such cases, e.g., security attacks against the system. Hence, the security properties that can specify dynamic systems should be employed. Besides, in some situations, it is difficult to predict and handle future security attacks or threats. Therefore, probabilistic approaches should be utilised as well. In this and the following sections, we introduce our language called GIZKA that allows us to express security properties of dynamic software systems. In this section in particular, we describe GIZKA.ESCL (also called ESCL which is subset of GIZKA) and its constructs for specifying alternative security properties and security properties that are time and event dependent and probabilistic.

GIZKA.ESCL extends SCL and involves time points, time intervals, time sequences, probability, alternative security properties, and runtime conditions.

Since GIZKA.ESCL employs various types of logic, we briefly introduce some works that we adopt. For example, probabilistic logic programming (PLP) is studied in [Luk01] where clauses are extended by a subinterval of [0,1]. An overview of modal logic, temporal logic, and interval logic is presented in [OM94] while time intervals and reasoning are studied in [HS91] and [BL02] accordingly. Other works which apply time in logic programming [BE02, AHK02, KV05] are not quite relevant to our approach. The issue of integrating logic programming and functional logic paradigms is studied in [Mar95] where a theory on how to formally express it and the functional core of logic programming are presented.

Time

In traditional security, time is used in security sessions or to set and check expiration of digital certificates and licenses, etc.

In GIZKA.ESCL, time is applied to express dynamic security properties and qualify their validity. We adapt coordinated universal time (UTC) as a reference point, which is measured from 1 January 1970 [Java06]. Normal day and time are used implying their translation to UTC.

In GIZKA.ESCL, we use two further types of time:

- \( T_a \) is the absolute system time, which is counted from the moment of time when the system starts working;
- \( T_r \) is the relative time which is the local time of a component and is counted from the moment of time when a component is launched.

As illustrated in Figure 6.3, \( T_a \) and \( T_r \) are linked in the following way: for a given moment, \( T_a = T_r + \Delta \) where \( T_a \geq 0, T_r \geq 0, \Delta \geq 0 \). If \( \Delta \) equals 0 then \( T_a \) equals \( T_r \). When \( T_a \) equals 0 it...

136
also equals Alpha in UTC time. Alpha represents the number of seconds that have passed since 1 January 1970 00:00:00.

(Time points)
Time point \(<T>\) describes exactly when security properties of components are valid. Time points are used to express time intervals and time sequences. For example, Alice and Bob know that the shared key is valid at 9am but there are no guarantees that it is still valid at other time periods. In GIZKA.ESCL, it can be expressed in the following way:

\[
\text{Shared(key,Alice,Bob),Time(<9am>).}
\]

(Time intervals)
Time intervals are employed to describe at which period of time certain security properties are valid. There are 14 types of time intervals, as shown in Figure 6.4.

- \(<T_1,T_2>\) - Security properties are valid exactly for the period of time from \(T_1\) to \(T_2\). For example, the shared key between Alice and Bob is valid from 9am to 5pm. In GIZKA.ESCL it can be expressed as:

\[
\text{Shared(key,Alice,Bob),Time(<9am,5pm>).}
\]
In fact, time interval can be considered as a security session. If we need security properties to be valid for a time duration $D$ without concerning the starting moment, it can be expressed in the following way: $<[D]>$, where $D$ is a time interval in seconds and $D=T_2-T_1$, where $T_2 \geq T_1 \geq 0$. For example, if the shared key between Alice and Bob is valid for 100 seconds, it can be expressed in GIZKA.ESCL as:

```
Shared(key,Alice,Bob),Time(<[100000]>).
```

- $<|T_1,T_2>|$ - Security properties are valid from $T_1$ to $T_2$ and they might be valid before $T_1$. For example, Alice and Bob know that the shared key is valid from 9am to 5pm, however, the key may or may not be valid before 9am:

```
Shared(key,Alice,Bob),Time(<|9am,5pm|>).
```

- $<|T_1,|T_2>$ - Security properties are valid at $T_1$. They also might be valid before $T_1$, after $T_1$ but before $T_2$. For example, Alice and Bob know that the shared key is valid at 9am and not valid after 5pm, but they are not sure that the key is valid before 9am, after 9am but before 5pm:

```
Shared(key,Alice,Bob),Time(<|9am,|5pm>).
```

- $<|T_1>$ - Security properties are valid at $T_1$ and not valid after $T_1$ but there is a possibility that they are valid before $T_1$. For example, Alice and Bob know that the shared key is valid at 9am and not valid after they also know that the key might be valid before 9am:

```
Shared(key,Alice,Bob),Time(<|9am>).
```

- $<||T_1>$ - Security properties might be valid before $T_1$ but they are not valid at $T_1$ or after $T_1$. For example, Alice and Bob may try to use the shared key before 9am but there are no guarantee that they will succeed:

```
Shared(key,Alice,Bob),Time(<||9am>).
```

- $<T_1,T_2|>$ - Security properties are valid for the period of time from $T_1$ to $T_2$ and they might be valid after $T_2$. For example, Alice and Bob know that the shared key is valid from 9am to 5pm; however, they are not sure that the key is valid after 5pm:
• \(<T_2\rangle\) - Security properties are not valid before \(T_2\), and valid exactly at \(T_2\), and there is a possibility that they are valid after \(T_2\). For example, Alice and Bob know that the shared key is not valid before 5pm, valid at 5pm, and there is possibility that the key is valid after 5pm:

\[
\text{Shared(key,Alice,Bob), Time(<5pm|>)}.
\]

• \(<T_2\|>\) - Security properties might be valid after \(T_2\) but they are not valid at \(T_2\) or before \(T_2\). For example, Alice and Bob may try to use the shared key after 5pm but there are no guarantees that they will be successful:

\[
\text{Shared(key,Alice,Bob), Time(<5pm||>)}.
\]

• \(<T_1|,|T_2>\) - Security properties might be valid after \(T_1\) and before \(T_2\) and they are not valid at any other periods of time. For example, Alice and Bob may try to use the shared key between 9am and 5pm but there are no guarantees that they will succeed:

\[
\text{Shared(key,Alice,Bob), Time(<9am|,|5pm>)}.
\]

• \(<|T_1,T_2|>\) - Security properties are valid between \(T_1\) and \(T_2\) and there are possibilities that they are valid before \(T_1\) and after \(T_2\). For example, Alice and Bob can use the shared key between 9am and 5pm; however, they may fail if they use it before 9am or after 5pm:

\[
\text{Shared(key,Alice,Bob), Time(<|9am,5pm|>)}.
\]

• \(<T_1|,T_2|>\) - Security properties are valid at \(T_2\) and they also might be valid before \(T_2\) but after \(T_1\), and after \(T_2\). For example, Alice and Bob know that the shared key is valid at 5pm and not valid before 9am, but they are not sure that the key is valid between 9am and 5pm or after 5pm:

\[
\text{Shared(key,Alice,Bob), Time(<9am,5pm|>)}.
\]

139
• $\langle | T_2 | \rangle$ - Security properties are valid at $T_2$ and they also might be valid before and after $T_2$. For example, Alice and Bob know that the shared key is valid at 5pm and they also might try to use it before or after 5pm:

\[
\text{Shared(key,Alice,Bob),Time($\langle |5pm| \rangle$)}.
\]

• $\langle T_1 |, T_2 \rangle$ - Security properties are valid for the period of time after $T_1$ to $T_2$ and they might be valid at $T_1$ but they are not valid before $T_1$ and after $T_2$. For example, Alice and Bob know that the shared key is valid after 9am to 5pm, however, there are no guarantees that the key is valid at 9am:

\[
\text{Shared(key,Alice,Bob),Time($\langle 9am|,|5pm\rangle$)}.
\]

• $\langle T_1,|T_2 \rangle$ - Security properties are valid exactly for the period of time from $T_1$ and before $T_2$ and they might be valid at $T_2$ but they are not valid before $T_1$ or after $T_2$. For example, Alice and Bob know that the shared key is valid from 9am to 5pm; however, there are no guarantees that the key is valid at 5pm:

\[
\text{Shared(key,Alice,Bob),Time($\langle 9am,|5pm\rangle$)}.
\]

(Time sequences)

Time sequences describe periods of time when security properties are valid. Members of a time sequence are ordered in the following sense: $\langle T_0 \rangle, \langle T_1 \rangle, \langle T_2 \rangle, \langle T_3, T_4 \rangle, \langle |T_5| \rangle, \ldots, \langle |T_{n-1}|, T_n \rangle$ where $0 \leq T_0 \leq T_1 \leq T_2 \leq \ldots \leq T_n$ and $n \geq 0$. For example, Alice and Bob can use the shared key from 9am to 10am and from 5pm till 7pm:

\[
\text{Shared(key,Alice,Bob),Time($\langle 9am,10am \rangle, \langle 5pm,7pm \rangle$)}.
\]

Probability

Probability is utilized if it is impossible to predict exactly what will happen with security properties of components, i.e., when these security properties will be valid. To describe probability, we employ an interval from 0 to 1 ([0,1]). A SCL expression $E_i \leftarrow R_i$ can be written with probabilistic properties as: $E_i([l_m,l_{m+1}]) \leftarrow R_i([p_k,p_{k+1}])$, where $p_k,p_{k+1},l_m,l_{m+1} \in [0,1]$ and $p_k \leq p_{k+1}$ and $l_m \leq l_{m+1}$. The interval $[0,0]$ means the security properties are not valid. The interval $[1,1]$ shows that the security properties are valid with 100% probability. For example, the shared
key between Alice and Bob is valid from 9am to 5pm with the probability from 80% to 90%. In GIZKA.ESCL it can be expressed as:

```
Shared(key,Alice,Bob),Time(<9am,5pm>),Probability([0.8,0.9]).
```

**Alternative security properties**

Alternative security properties (in contrast to individual properties) can be formed in blocks in order to allow the component requestor to select an alternative match from a set of possible properties: \{E_i←R_i,E_j←R_j,E_l←R_l\}, \{E_m←R_m,E_n←R_n\},…, where \(i ≥0, j ≥0, l ≥0, m ≥0, n ≥0\). Security property sets are shown as \{\ldots\}. Only one of the individual security properties can be chosen from each set. A set of properties with one property inside \(E_i←R_i\) can be written as \(E_i←R_i\).

**Runtime conditions**

Runtime conditions express when and how security properties are valid. They allow describing flexible scenarios:

```
IF (<conditions>) {<security properties>}
ELSE {<security properties>}. 
```

**Representing security properties**

Some of security properties for cryptographic functions, which we use, are presented below:

- **KeyExchanged(K,P,Q),Time(<T_i,T_{i+1}>,),Probability([p_k,p_{k+1}]):** K is exchanged between P and Q at time between \(T_i\) and \(T_{i+1}\) with probability between \(p_k\) and \(p_{k+1}\) where \(i ≥0, k ≥0\).
- **BelievesShared(P,Q,X),Time(<T_i,T_{i+1}>,),Probability([p_k,p_{k+1}]):** P believes that it shares X only with Q at time between \(T_i\) and \(T_{i+1}\) with probability between \(p_k\) and \(p_{k+1}\) where \(i ≥0, k ≥0\). P can decrypt and see X if it is encrypted by Q with a shared key K.
- **KeyDistribution(K,A,S),Time(<T_i,T_{i+1}>,),Probability([p_k,p_{k+1}]):** K’s distribution algorithm is A and standard is S at time between \(T_i\) and \(T_{i+1}\) with probability between \(p_k\) and \(p_{k+1}\) where \(i ≥0, k ≥0\).
- **Shared(K,P,Q),Time(<T_i,T_{i+1}>,),Probability([p_k,p_{k+1}]):** a key K is shared between P and Q at time between \(T_i\) and \(T_{i+1}\) with probability between \(p_k\) and \(p_{k+1}\) where \(i ≥0, k ≥0\).

If a security property does not include probability it means that probability equals 1 or \([1,1]\). If time is not used in a security property it means that the property is not time-dependent. Moreover, if time or probability is placed after the security properties, it means that they relate to the whole block of those properties.
Compatibility reasoning
The required and ensured security properties of a service requestor (F) and responder (C) need to be compatible relative to time and probability before F can invoke C.

(Time reasoning)
When the security properties of F and C are valid with probability 100%, F can invoke C if their time properties concur exactly. For example, if F requires a service from 9am to 5pm and C provides it from 8am to 5pm then F can invoke C because time periods concur. If F requests a service from 9am to 10am and C provides this service from 11am till 4pm, then time periods do not intersect and F cannot invoke C. If F requests a service from 9am to 11am and C provides it from 10am to 4pm then F can try to find other responders which satisfy F’s requirements or invoke C from 10am till 11am (and find other responders which provide services from 9am to 10am) because there is only a partial intersection of time periods from 10am to 11am and the requirements are not fully met.

(Probability reasoning)
Let us assume that time periods in the security properties of F and C concur exactly. When F requires a service from C with probability \([p_1,p_2]\) and C provides this service with probability \([p_3,p_4]\), F can invoke C if \(p_3 \geq p_1 \geq 0\) and \(p_4 \geq p_2 \geq 0\) where \(p_2 \geq p_1\) and \(p_4 \geq p_3\). For example, F requests a service with probability 80%-90% and C provides it with probability 85%-95%. In this case, F invokes C because C guarantees better probability. In the case if C provides a service with lower probability (e.g., 50%), a service can still be used if F agrees to lower its probability requirements, otherwise, F should find another service provider.

Summary
It is necessary to mention that time and probability is very useful to state when certain services can be available and with which probability allowing the requestor to choose a proper component with a required level of availability. Besides, they can be utilised during the risk assessment process. For example, to build the risk model, the assessor can use time and probability, i.e., certain services can be available at the certain period of time and have certain protection with certain probability.

6.2.4 Additional constructs for specifying security properties
In this section, we present additional constructs used to define security properties of components or services.

Types
GIZKA.ESCL is a typed language similar to Java 6 [Java06] and supports three integer types (LONG, INTEGER, and BYTE), two floating types (FLOAT and DOUBLE), and three special floating-point values to mark overflows and errors including positive infinity, negative infinity, and NaN (not a number). Besides, GIZKA.ESCL supports strings that are sequences of Unicode characters and defined by the type STRING. The BOOLEAN type is employed to evaluate logical conditions and has two values, FALSE and TRUE. The types TIME and DATE are represented by the class ‘Time’ from our security ontology described previously.

Variables and constants
In GIZKA.ESCL, variables are represented by the keyword VAR. A software developer declares a variable by putting VAR and the type first and then the name of the variable. The constants are denoted by the keyword FINAL in front of the type and name of the variable indicating that the variable can be assigned only once. Moreover, software developers can create complex data types using the keyword STRUCT. Finally, by using the keyword ALIAS developers can specify aliases for a particular variable or constant to utilise shorter names.

Operators
GIZKA supports the relational and boolean operators and the arithmetic operators (similar to Java and C++).

Other constructs
The keyword ENUM is used in cases when a variable holds a restricted set of values. The keyword ARRAY denotes a data structure used to store a set of values of the same type. The keyword IN or OUT is placed in front of every variable that goes as a parameter for a function.

Program blocks
The security interface consists of program blocks as illustrated in Figure 6.5.

```plaintext
INTERFACE Name {
    #declaration of variables and constants
    VAR {...}
    #functionality block: declaration of required and provided functionalities
    FUNCTIONALITY {
        #declaration of required functionalities
        REQUIRED {...}
        #declaration of provided functionality
        PROVIDED {...}
    }
    #security block: declaration of security properties (requirements and capabilities)
    SECURITY { REQUIRED {...} PROVIDED {...} }
}
```

Figure 6.5. Program block
The declaration of security interface of a component starts with the keyword **INTERFACE** followed by the name of the component. Then, the block containing variables and constants starts with the keyword **VAR** where data structures, different variables and constants are declared. Other high level blocks are **FUNCTIONALITY** and **SECURITY**. **FUNCTIONALITY** block specifies functionalities provided (PROVIDED) or required (REQUIRED) by the particular component or service. The block with the keyword **SECURITY** declares the ensured and required security properties for different functionalities. The keyword **PROTECTED** in front of the keywords **PROVIDED** or **REQUIRED** means that the particular block is only accessible to components which are members of the particular domain. The keyword **SELF** denotes to the component itself.

**Example**

In this section, we demonstrate how program blocks can be used. We specify an interface of a certification/key distribution component (KDC) described in Chapter 3. This KDC provides several functionalities including `getSharedKey`. Security properties involved with functionality `getSharedKey` provided by this KDC employ IKE, Diffie-Hellman, PGP, and 3DES algorithms. The key size of the shared key is 168 bit. The whole example is illustrated in Figure 6.6.

```plaintext
INTERFACE KDC {
    VAR {
        SASO.SecurityAlgorithm.KeyExchange.IKE ALIAS IKE;
        SASO.SecurityAlgorithm.KeyExchange.DeffieHellman ALIAS DiffieHellman;
        SASO.SecurityAlgorithm.SAlgEncryption.SAlgSymmetric.3DES ALIAS 3DES;
        SASO.SecurityConcept.SConProtocol.SConEmail.PGP ALIAS PGP;
        /*The “ENUM” contains information about various security algorithms*/
        ENUM Algorithm {IKE,DiffieHellman,PGP,3DES};
        /*Variables K introduces a shared cryptographic key*/
        SFO.SecurityEntity.KEY.SymmetricKey ALIAS K;
        INTEGER L=168;
    }
    FUNCTIONALITY {
        REQUIRED {...}
        PROVIDED {
            STRING getSharedKey(IN Services.Client c, IN Services s, IN Algorithm a);
        }
    }
    SECURITY {
        PROVIDED getSharedKey(Services s, Algorithm a) {
            KeyAlgorithm(K[SELF,s],a.3DES,a.PGP).
            KeySize(K[SELF,s],168).
            KeyGenerated(K[SELF,s])←KeyAlgorithm(K[SELF,s],a.3DES,a.PGP),KeySize(K[SELF,s],L).
            Shared(K[SELF,s],SELF,s)←KeyGenerated(K[SELF,s]).
            KeyDistribution(K[SELF,s],a.IKE,a.DiffieHellman).
        }
    }
}
```

Figure 6.6. Interface example
In the block starting with the keyword VAR we firstly declare constants and variables used in the security block including various security algorithms, the key size, and the key itself. Then, in the functionality block we state that this component provides security functionality `getSharedKey`. Finally, in the block security we specify the security properties related to the provided functionality `getSharedKey`.

### 6.2.5 Reasoning

In this section, we describe how software components or services can negotiate and verify if one’s security requirements are satisfied by the other’s security capabilities and vice versa. However, the process of reasoning cannot be fully automated and still needs the human intervention (e.g., a system administrator). We divide this process into several steps including reasoning about functionality, probability and time properties, and security properties to make the process of reasoning semi-automatic and improve performance. This process is illustrated schematically by the flowchart in Figure 6.7.

![Figure 6.7. Reasoning](image-url)
Reasoning rules

The first step of the process of reasoning is to check all functionalities. During this process a component requestor (REQU) ensures that a component responder (RESP) provides needed functionality. If matching was not successful on the first step and there are no matches then reasoning is stopped. Otherwise the process of reasoning goes to the next step that includes verifying probability properties using the probabilistic reasoner (introduced in Chapter 8). If REQU requires a service from RESP with probability \([p_1,p_2]\) and RESP provides this service with probability \([p_3,p_4]\), then a match occurs if \(p_3 \geq p_1 \geq 0\) and \(p_4 \geq p_2 \geq 0\) where \(p_2 \geq p_1\) and \(p_4 \geq p_3\). If there is no match then negotiation is finished, otherwise this step leads to the next step on which time properties are verified by the time reasoner (described in Chapter 8). More specifically, REQU can require support for certain security functionality at certain periods of time while RESP also can provide this security functionality at certain periods of time. Several cases are possible:

- If time periods of REQU and RESP concur exactly then a match occurs.
- If the time period of RESP is greater than REQU’s time period then a match occurs.
- If the time period of REQU is greater than RESP’s time period then no match occurs.
- If the time periods of REQU and RESP partially intersect then no match occurs.

If there is no match then the process of reasoning is terminated otherwise it goes to the next step which is verifying security properties for the particular functionality chosen previously where reasoning rules such as perfect match, close match, possible match, and no match in decreasing order of matching are employed. This step is partially based upon [KLK05]. Since the security properties are expressed in logic programming then we follow the approach proposed in [Kha05] where a reasoning engine adopts a tool called smodels [NS00]. Smodels is an inference engine aimed at computing answer sets (stable models) of programs of A-Prolog [NS00,EFL00] that can deal with time, time intervals [SBT04,GGS+04,Jon06], and probability. Smodels performs actual reasoning and is usually utilised together with another tool called lparse [Syr00] that syntactically prepare security properties for smodels. We present the detailed description of the architecture of our reasoning engine in Chapter 8.

Perfect matches can happen if one’s capability and the other’s requirement point to the same concept or two concepts declared as equivalent in the ontology. More specifically, the requirement and capability point to the same ontological concept (e.g., REQU needs the AES algorithm with the key size 256 and RESP provides it) or the requirement and capability specify equivalent concepts in the ontology.

Close matches occur if one’s requirement is less detailed than the other’s capability. Three cases are possible including the requirement points to a more general concept (higher level) in the ontology than the capability; the requirement and capability point to the same concept, however,
the capability is described in more details; and the requirement is specified in terms of security objectives while the capability is described in terms of security concepts that support these objectives.

Possible matches occur when one’s requirement is defined in more details than the other’s capability. Similar to close matches, there three cases that can result in possible matches including the requirement specifies a more specific concept that lies lower in the ontology; the requirement and capability refer to the same concept, however, the requirement is expressed in more details; and the requirement is defined in terms of security concepts while the capability is stated in terms of security objectives.

No matches happen if one’s requirement and the other’s capability are fully incompatible. The cases leading to no possibility of matching are when the requirement and capability specify two unrelated concepts and the requirement and capability point to the same concept but have different concept properties. If a no match occurs then the process of reasoning stops. Perfect matches, close matches, and possible matches lead to a successful match. It is necessary to mention that on each step there are possibilities to negotiate better choices.

Example
In this section, we introduce an example (partially extracted from [KLK05]) of the process of reasoning.

(Introduction)
For example, there is a component requestor (REQU) that is looking for a component that provides a book flight ticket service (BookFlightTicket functionality). Besides, REQU requires that a component provider should satisfy certain security requirements such as authorisation and support of SSH and AES security algorithms. Also, REQU provides some capabilities such as authentication via SAML algorithm that supports X.509 certificates signed by VeriSign. REQU requires that this functionality and security properties should be supported from 10am till 5pm every day with the probability from 90% to 95%.

Suppose, there is a component provider (RESP), that delivers BookFlightTicket functionality and supports certain security properties. However, RESP guarantees only with probability more than 95% that this service will be available on the 24/7/365 basis. If REQU is satisfied with such capabilities then it can use RESP, otherwise, it tries to find another component provider. RESP’s security capabilities include support of SSH with AES and authorisation through VPN.

At the same time, RESP requires authentication from other components via SAML with X.509 certificates.
Below, we introduce **REQU**’s and **RESP**’s requirements and capabilities.

**REQU**’s functional requirements:

//booking flight tickets functionality.
BookFlightTicket.

**REQU**’s security capabilities:

//Authenticity via SAML with X.509 certificates signed by VeriSign.
SASO.SecurityObjective.Authenticity ALIAS Authenticity.
SASO.SecurityConcept.SConProtocol.SConAuthentication.SAML ALIAS SAML.
Authenticity←Signed(X509Certificate,VeriSign).

To reduce the amount of source code, the first three lines declare constants used in the security properties. The last line states that authentication is ensured if X.509 certificates signed by VeriSign are used. Other requirements and capabilities are discussed and analysed below.

**REQU**’s security requirements:

//Authorisation and SSH with AES.
SASO.SecurityObjective.Authorisation ALIAS Authorisation.
SASO.SecurityConcept.SConProtocol.SConCommunication.SSH ALIAS SSH.
SASO.SecurityAlgorithm.SAlgEncryption.SAlgSymmetric.AES ALIAS AES.
←Authorisation.
←KeyAlgorithm(K[REQU,RESP],SSH,AES).
//time requirements
//BookFlightTicket functionality with security properties should be supported from 10am till 5pm every day.
←Time(<10am,5pm>).
//probability requirements
//Probability should be from 90% to 95%.
←Probability([0.9,0.95]).

**RESP**’s functional capabilities:

BookFlightTicket.

**RESP**’s security requirements:

//Authenticity via SAML with X.509 certificates
SASO.SecurityObjective.Authenticity ALIAS Authenticity.
SASO.SecurityConcept.SConProtocol.SConAuthentication.SAML ALIAS SAML.
←Authenticity.

//Authorisation via VPN and SSH with AES.
SASO.SecurityObjective.Authorisation ALIAS Authorisation.
SASO.SecurityConcept.SConProtocol.SConCommunication.SSH ALIAS SSH.
Chapter 6  GIZKA: a language for managing security

In this section, we demonstrate the process of reasoning. Firstly, **REQU** needs to find a component provider with the desired functionality ‘BookFlightTicket’. **RESP** provides this functionality, hence, the process of reasoning goes to the next step that is verifying probability properties. Because **RESP** provides better probability than **REQU** requires, hence, probability properties are also satisfied. The next step of the process of matching goes to verifying time properties. Since **REQU** requires the desired functionality and security only at certain periods of time while **RESP** provides them all the time, hence, the time periods are fully overlapped. Then, the next step is reasoning about security properties. **REQU**’s security capabilities satisfy **RESP**’s security requirements with the close match because the first **RESP**’s requirement and the first **REQU**’s capability point to the same concept but **REQU**’s capability has more details. On the other hand, **REQU** requires authorisation that is provided by **RESP** through VPN. Besides, both **REQU** and **RESP** support SSH and AES, hence, there is the perfect match. However, the lowest level of match is the close match, therefore, this is the final step which finishes the process of reasoning with successful results.

6.3 Description of attack correlation and anti-correlation

In the previous chapters, we introduced our security ontologies that are utilised in GIZKA.ATTACK (subset of GIZKA) and briefly demonstrated through the use of the Mitnick attack how these ontologies can be adopted in detecting multi-phased distributed attacks. Actually, the intrusion detection system (IDS) research community assumes the fact that there is no a silver bullet defence against various security attacks and the best solution is to develop a multi-layered perimeter defence with various intrusion detection and prevention systems.

In this section, we describe in more details how security attack correlation and anti-correlation (i.e., a selection of required countermeasures against a security attack) can be done and present a subset of GIZKA features that are used to define security attack correlation and anti-correlation. More specifically, the process of correlation and anti-correlation can be subdivided into three stages:

- Identifying independent security attacks that are possibly related to each other;
• Analysing the root attack that may trigger independent security attacks and trying to predict the attacker’s goal;
• Performing anti-correlation, i.e., developing and deploying required countermeasures (e.g., system’s reconfiguration as presented in the next section).

6.3.1 Introduction to attack languages

Every security attack can be defined by a number of unique parameters that form a unique attack signature. In order to describe such signatures in a way understandable by software agents as well as humans, several attack languages were developed.

In this section, we present attack languages used to encode descriptions of security attacks in a suitable format (e.g., attack signature), then recognise these attack descriptions, and react or report about them [VEK00]. Attack languages can be very useful for analysing relationships among different attacks (seem to be independent) in order to detect more complex security attacks such as multi-phased distributed attacks against a system. For example, STATL [EVK02] is a domain-independent attack description language that can be extended to match different environments. STATL supports misuse detection and specifies attack scenarios in a domain-independent way. This language was utilised in defining network-based and host-based security attacks and an implementation of a toolset of intrusion detection systems based on STATL was developed as well.

According to [VEK00], attack languages can be subdivided into six different language classes:
• Event languages – used to describe events and mainly focus on the specification of data format. Examples are tcpdump packets or syslog messages for UNIX-like operating systems [WIKI07].
• Reporting languages – employed to describe alerts that contain information (e.g., the source of the security attack, the target of the attack, etc.) about security attacks. It is worth mentioning that reporting languages can be used as event languages. Examples are the Common Intrusion Specification Language (CISL) which is part of the Common Intrusion Detection Framework (CIDF) [CIDF99] and the Intrusion Detection Message Exchange Format (IDMEF) [IDMEF04].
• Correlation languages – rely on a large number of alerts from different intrusion detection systems (IDSs) and try to recognise the whole picture of the attack, i.e., try to identify relationships among different security attacks in order to detect coordinated complex attacks. STATL [EVK02,Eck01] is an example of the correlation language.
• Exploit languages – utilised to specify steps needed to perform a security attack. For example, NASL (Nessus Attack Specification Language) [NASL07] provides language-level support for attack scripting or ADeLe (Attack Description Language) [MM01].
• Detection language – specially developed to support intrusion detection through specifying attack signatures and usually referred as attack languages. Examples are Snort rules used in the Snort intrusion detection/prevention system [Snort07], STATL [EVK02] employed in the STAT Toolset, LAMBDA (Language to Model a Database for Detection of Attacks) [CO00], and ADeLe [MM01] designed to model a database of known attack scenarios.

• Response languages – used to specify actions to be taken in response to the registration of attack signatures. Currently, there is no a well-defined response language, just a number library functions written in such languages as C or Java for specific needs. In the next section, we present our response language which is a subset of the GIZKA language.

However, it is worth mentioning that there is no a standardized language for specifying security attacks and defences and there is no a common ontology that would allow security events to be conceptualized and codified in a standardized way.

6.3.2 Attack identification

According to [MCA01], there are two attack perspectives: the victim's view and the attacker's view which focus on the following manifestations [Und04]:

• Victim’s view:
  o What happened?
  o Who and what is affected?
  o Who is the attacker?
  o When and where the attack is originated?
  o Why and how the attack happened?

• Attacker’s view:
  o What is the attacker’s objective?
  o What vulnerabilities exist and can be used in the target system?
  o What damage or other consequences are likely?
  o What attack tools are available?
  o What is attacker’s risk of exposure?

Since we are “good guys”, we follow the victim’s view in developing our attack language. However, it is also possible to apply the attacker’s view through extending security ontologies. In order to identify security attacks and describe them in details, we also need an attack language. We can create a new language and specify all attacks by ourselves or we can choose an existing one. In the first case we get more flexibility while in the second case we have more constraints. However, we mix both cases, i.e., we develop our own language called GIZKA.ATTACK that is a subset of GIZKA and codifies information regarding security attacks for the security attack ontology (SAO) and employ existing attack signatures through converting
them into GIZKA.ATTACK. For such purpose, we employ data related to computer attacks, vulnerabilities and exploits gathered from multiple sources including (but not limited to) a common vulnerability and exposure (CVE) dictionary provided by the MITRE corporation [CVE07], the Milw0rm website [Milw0rm08], the Microsoft website [Microsoft07], the Security Focus [SF07], and Snort rules [Snort07] and some others. There are about 14000 Snort rules that are free and regularly updated. Actually, Snort is a tool that is capable of performing real-time traffic analysis and packet logging on IP networks. Snort is a signature-based intrusion detection system (IDS) that consists of a sniffer, preprocessors, a detection engine and postprocessors. Snort utilises configuration files including an actual configuration file and files containing a large collection of published Snort attack signatures. For example, the class ‘TeardropAttack’ introduced in the previous chapter can be specified in Snort in the following way, as depicted in Figure 6.8:

```
First file
```

```
Second file
Rule:--Sid:270--Summary:This event is generated when an attempt is made to issue a Teardrop Denial of Service (DoS) attack.
--Impact:Denial of Service.--Detailed Information:Teardrop exploits a vulnerability in some TCP/IP stack implementations. The program sends a specially crafted fragmented packet where the first fragment has offset 0 and data length N and the second fragment has an offset less than N (The fragments overlap). The resulting packet cannot be properly assembled. Systems may hang or crash.
--Affected Systems: Windows 95 Windows NT 4.0 SP3 and earlier HP HPUX 10.34 and earlier Linux kernels 2.0.31 and earlier FreeBSD 3.0 prior to October 27, 1998
--Attack Scenarios:The can be done remotely against any open UDP port using a spoofed address.
--Ease of Attack:Simple. Tools are readily available and require little knowledge on the part of the attacker.
--False Positives:None known.--False Negatives:None known.
--Corrective Action:Patches are available from all affected vendors. Newer versions from each vendor are not vulnerable.
--Contributors:Original Rule Writer Unknown Sourcefire Research Team Nigel Houghton <nigel.houghton@sourcefire.com> Snort documentation contributed by Steven Alexander<alexanders@mccd.edu>
```

Figure 6.8. Snort rule

This rule says that the event with Sid:270 is generated in the case of the detection of a Teardrop DoS attack which exploits a vulnerability in some TCP/IP stack implementations. As previously mentioned in Chapter 5, an attacker sends a specially crafted fragmented packet where the first fragment has offset 0 and data length N while the second fragment has an offset less than N resulting in the fragments overlap. Hence, the resulting packet cannot be properly assembled, and consequently, the system under attack may freeze or crash.

Every Snort rule has a rule header and rule options. The rule header consists of the rule action, the protocol, the source and destination IP addresses with netmasks, and the source and destination ports. The rule options contain messages to be displayed or logged for matching packets, predicates on packet fields, and response directives.
It is necessary to mention that the rule in Figure 6.8 should be on a single line and that in this Figure we depict two files, the part of the file “dos.rules” (contains the actual attack signature) and the whole file “270.txt” (has the additional information related to the attack such as affected systems and attack scenarios). First, the rule header starts with the keyword ‘alert’. This rule matches UDP packets from any external source IP address and port to any port on the local network. Rule options state that this rule specifies “DOS Teardrop attack” with fragbits:M. Other rule options refer to other descriptions of this attack in other sources such as Bugtraq or CVE and sid:270. The detailed description of Snort rules can be found in [Snort07] while the example of how to convert Snort rules to STATL [EVK02] is presented in [Eck01]. Moreover, it is worth mentioning that Snort has various limitations. It is rather difficult to write a complete Snort parser because its syntax and semantics are not fully documented [SP03]. Besides, Snort has about 14000 signatures in Snort version 2.8 and about 4000 in Snort version 2.4 by default, however, the default configuration generates too many false positives. Moreover, there are some overlapping signatures for the same security attack and some of the signatures are not correct.

6.3.3 Attack correlation and anti-correlation

After security attacks (that may seem to be independent) have been identified, they should be correlated in order to find the root attack. To clarify, correlation is the process of piecing together various security events that are related to each other. However, classical intrusion detection systems detect only elementary attacks while current security attacks tend to be multi-phased and distributed. Hence, there is the need for the correlation function that allows identifying complex attacks from the set of simple attacks, and even predicting next steps of the attacks from the early steps. Besides, security attacks should be anti-correlated, i.e., proper countermeasures should be chosen. There are two main correlation approaches [Mar04]:

- Explicit correlation of events – a human (e.g., a system administrator) can recognize some correlations among events using her knowledge of relations among alerts or employing dependencies of the topology of system’s components;
- Implicit correlation of events – some mappings and relations among events are generated using data analysis.

The detailed description of attack correlation for cooperative intrusion detection can be found in [Und04] or in [CM02] where LAMBDA (Language to Model a Database for Detection of Attacks) [CO00] is utilised to specify security attacks and countermeasures.

In [CGS04] a LAMBDA attack description consists of several attributes:

- Pre-condition – describes the state of the system needed for the attack to be successful;


- Post-condition – specifies the state of the system after the successful occurrence of the security attack;
- Detection – is the description of the alert that corresponds to the identification of the attack;
- Verification – defines the conditions to verify if the attack was successful.
- State (intrusion objective) – describes the state of the system that relates to violation of security policies

A LAMBDA defence description consists of the same attributes (except that action replaces detection) [CGS04]:
- Pre-condition – describes the state of the system needed for the countermeasure to be successful;
- Post-condition – specifies the state of the system after the successful applying of the countermeasure;
- Action – expresses required actions to apply the countermeasure;
- Verification – defines the conditions to verify if the countermeasure was successful.

According to [CGS04], once all security attacks are specified, it is possible to generate correlation unifiers between attacks. Also, if the first step of the attack is identified it is possible to predict next steps and generate the knowledge about future attack steps and possible intrusion objectives. Such attacks are called virtual attacks. They allow to apply a countermeasure even in the beginning of the attack scenario instead of deploying it after the attack occurred.

The idea is to select a “right” countermeasure (called anti-correlation [CGS04,CAB+06]) that would be effective against a particular attack is not new. Direct responses against an attacker can be a rather complex problem from the technical perspective (the need to know the precise source of the attack) as well as from the ethic and legal perspective. Similar to [CAB+06], we focus only on the technical issues.

### 6.3.4 GIZKA.ATTACK

GIZKA.ATTACK supports several constructs that allow to identify attacks, correlate and anti-correlate them. We present them below using the example of the Mitnick attack from the previous chapter. It is necessary to say that the security defence ontology (SDO) specifies countermeasures and is codified using GIZKA. Besides, SDO correlates with the security attack ontology (SAO) that uses attack signatures specified in GIZKA as well. The problem is how to choose an appropriate defence against a particular attack. It can be done by a human administrator or by an artificial intelligence agent [Mar04]. A proper mapping (correlation) among various possible security events (e.g., security attacks) or combination of them and the root security attack that triggered them is not straightforward and usually requires a human security expert to resolve this issue. Selecting a proper countermeasure (anti-correlation)
Automating both tasks requires to employ artificial intelligence techniques that go beyond usual rule-based systems that simply use the knowledge of the security expert about the independence of security events. The main drawbacks of rule-based systems are [HMP02]:

- It is practically infeasible to list the complete set of rules beforehand in complex domains;
- It is difficult to adapt them to environment changes;
- They require constant maintenance, hence, the process of keeping up to date and writing rules is time-consuming, challenging, and error-prone;
- Human experts are required to subsequently verify their outcome.

Case-Based Reasoning has been proposed in [Mar04] to solve the issues of rule-based systems utilised for intrusion detection to identify multi-staged attacks and multi-goals attacks. Analysis and decisions are performed by the manager who can be a human (e.g., security expert) or a software agent.

To demonstrate GIZKA.ATTACK, we use the example of the Mitnick attack from the previous chapter. Besides, as mentioned in the previous chapter, a security attack can trigger security events expressed by the class ‘SecurityEvent’ from the security attack ontology (SAVO). The following class has a number of properties including ‘NAME’, ‘TARGET’, ‘SOURCE’, and ‘TIME’. This class and its data properties are specified as follows and used by GIZKA.ATTACK as a main language concept to define security attacks, and consequently, identify, correlate and anti-correlate them:

```java
Class SAVO.SecurityEvent {
    Property NAME;
    Property TARGET;
    Property SOURCE;
    Property TIME;
}
```

Properties ‘NAME’, ‘TIME’, ‘TARGET’, and ‘SOURCE’ describe what type of security event has occurred, during what period of time, which IP address has been targeted, and from which IP address it has been initiated. If the attack occurs, it triggers a security event represented by the instance of the class ‘SecurityEvent’ (specified by the keyword ‘NEW’ and executed using the keyword ‘EMIT’) called ‘SecEv1’ defined as follows:

```java
SAVO.SecurityEvent SecEv1 = NEW SAVO.SecurityEvent {
    NAME = SAO.Attack.DoSAttack.DDoSAttack.TCPSYNAttack;
    TARGET = 147.202.46.43;
    SOURCE = 147.202.46.40;
    TIME = 14.40.00 07.07.2007 - 14.45.00 07.07.2007;
}
EMIT SecEv1;
```
It states that too many connections have been opened from the IP address ‘147.202.46.40’ from 14.40.00 to 14.45.00 (5 minutes) on the 7th of July 2007 with the host H1 (the IP address equals ‘147.202.46.43’) (see the previous chapter). Another event ‘SecEv2’ is generated using the keywords ‘NEW’ and ‘EMIT’ when H2 detects that someone from the IP address 190.2.1.23 tries to predict a TCP sequence number. This event is specified as:

```
SAVO.SecurityEvent SecEv2 = NEW SAVO.SecurityEvent {
  NAME = SAO.Attack.TCPSequenceNumberPredictionAttack;
  TARGET = 147.202.46.40;
  SOURCE = 192.2.1.23;
  TIME = 14.40.00 07.07.2007 - 14.45.00 07.07.2007;
}
EMIT SecEv2;
```

Then, these two events are correlated using rules, as described in Chapter 5. The conditional clauses such as ‘IF-ELSE’ are utilised to describe these rules:

```
IF(SecEv1.NAME == SAO.Attack.DoSAttack.DDoSAttack.TCPSYNAttack AND
  SecEv2.NAME == SAO.Attack.TCPSequenceNumberPredictionAttack
  AND SecEv1.SOURCE == SecEv2.TARGET AND SecEv1.TIME == SecEv2.TIME) {
  SAVO.SecurityEvent SecEv3 = NEW SAVO.SecurityEvent {
    NAME = SAO.Attack.MultiPhasedDistributedAttack.MitnickAttack;
    SOURCE = 192.2.1.23;
    TIME = 14.40.00 07.07.2007 - 14.45.00 07.07.2007;
  }
  EMIT SecEv3;
}
```

Moreover, events triggered by XML injection (the class ‘XMLInjectionAttack’) and WS Probing attacks (the class ‘WSProbingAttack’) and identified by H2 can be defined and emitted as follows:

```
SAVO.SecurityEvent SecEv4 = NEW SAVO.SecurityEvent {
  NAME = SAO.Attack.WSAttack.DiscoveryAttack.WSProbingAttack;
  SOURCE = 192.2.1.23;
  TARGET = 147.202.46.40;
  TIME = 14.40.00 07.07.2007 - 14.45.00 07.07.2007;
}
EMIT SecEv4;
SAVO.SecurityEvent SecEv5 = NEW SAVO.SecurityEvent {
  NAME = SAO.Attack.WSAttack.XMLInjectionAttack;
  SOURCE = 192.2.1.23;
  TARGET = 147.202.46.40;
  TIME = 14.40.00 07.07.2007 - 14.45.00 07.07.2007;
}
EMIT SecEv5;
```

The manager correlates all these events and decides that they are related to each other and performed by one attacker. Then, the manager labels the combination of these attacks as the WS Mitnick attack and sends a description of the attack using SAO to other hosts including H1, H2, H3, and H4 (see the previous chapter).
As previously mentioned, these attacks should be treated as a part of coordinated multi-phased distributed attacks and not as independent attacks, i.e., they should be analysed (e.g., by the manager) as a part of one complex attack in order to see, control, and maintain the whole picture of attacks. Such approach helps to develop more comprehensive countermeasures faster.

Countermeasure against WS Mitnick attacks are specified by the class 'WSMitnickDefence' from the security defence ontology (SDO) which contains an array of actions (the keyword 'ACTIONS') including closing all connections from the attacker’s IP address and blocking this address, increasing the size of the connection queue and decreasing the time-out waiting for the three-way handshake. The described constructs as the part of GIZKA.ADL are presented in the next sections while some other constructs are illustrated below where the keyword 'ENFORCE' is used to enforce countermeasures:

```plaintext
SAVO.SecurityEvent SecEv6 = NEW SAVO.SecurityEvent {
NAME = SAO.Attack.MultiPhasedDistributedAttack.WSMitnickAttack;
...
}
EMIT SecEv6;
IF(SecEv6.NAME == SAO.Attack.MultiPhasedDistributedAttack.WSMitnickAttack) {
SDO.WSMitnickDefence WSMitnickDefence_1 = NEW SDO.WSMitnickDefence {
ACTIONS {
CLOSE_CONNECTIONS_IP=147.202.46.40;
BLOCK_IP = 192.2.1.23;
INCREASE_CONNECTION_QUEUE = +50;
DECREASE_TIMEOUT = -1ms;
}
}
ENFORCE WSMitnickDefence_1;
}
```

If H1 detects the WS Mitnick attack, it closes all connections from the spoofed IP address (147.202.46.40), then increases the size of the connection queue, decreases the time-out waiting for the three-way handshake, and blocks all connections from the A’s IP address (192.2.1.23).

The implementation details of GIZKA.ATTACK are presented in Chapter 8.

### 6.4 Structural reconfiguration as a countermeasure

After security attacks have been detected, analysed, and correlated the system should respond. According to [GD02], responses can be subdivided into four categories:

- **Information** – an action that raises an alert such as a warning message for the system administrator.
- **Deterrence** – an action that forces an attacker to stop her malicious activity. For example, a warning message sent to the attacker stating that the malicious activity is detected and recorded.
• Correction – an action that changes the system state in order to correct a vulnerability used by an attacker. For example, the system patching can be considered as a mechanism of correction.

• Compensation – an action that blocks, and consequently stops the attack but without correction. The system stays vulnerable because a mechanism of correction has not been applied yet but the security attack has been already stopped. For example, the system can block the connection with the attacker or reconfigure itself structurally in order to survive during the attack.

Structural reconfiguration and adaptation can be treated as the part of correction and compensation, hence, we focus on these two categories. However, other categories such as information and deterrence are considered as well.

Also as previously mentioned, there is no fundamental defense against some subclasses of DoS attacks including DDoS and DRDoS (e.g., Smurf or Fraggle) because of their nature. For example, since DRDoS attacks exploit normal functionalities of the Internet, it is not possible to resist attacks by traditional countermeasures such as blocking packets from the DNS servers or other key Internet services. If the victim does it, she blocks herself from key sections of the Internet. Moreover, there are a few other traditional defenses against such attacks including filtering UDP and ICMP packets, disabling directed broadcast in all network broad routers, dropping ICMP ECHO packets, or ignoring ICMP broadcasts entirely (the radical method). In our approach, we propose to use structural reconfiguration and adaptation of the system as an additional element to resist and mitigate these security attacks.

However, to do it effectively, these reconfigurations should be specified in a formal way. For such purpose, we have developed and introduce in this section a language GIZKA.AD (a subset of GIZKA) and demonstrate it through the use of various examples.

### 6.4.1 Architecture description languages

The architecture description language (ADL) is a language for describing software architectures that enables developers to abstract away unnecessary low-level details, focus on the high level system structure and communication protocols, assign software components, etc. [Med97] classified and compared a number of ADLs including Aesop [GAO94], C2 [MRT99], Darwin [MDE+95,Jac99], MetaH [BEJ+96], Rapide [LKA+95], SADL [MR97], UniCon [SDZ96], and Weaves [GQ94]. The more recent ADLs are Others in ACME [GMW00,ACM05], Wright [ADG98,EHK05], and xADL and Secure xADL [RTD+05]. For example, Rapide is treated as an ADL and a simulation language at the same time while Wright is employed to formalize semantics of architectural connections. ACME focuses for architectural interchange, mainly at the structural level. However, none of current ADLs support the description of dynamic structural reconfigurations and adaptations together with security properties in order to achieve
higher-level security. We have selected ACME [GMW00] for its fundamental capabilities: 1) Architectural interchange, 2) Extensible foundation for new design and analysis tools, 3) Architecture description. ACME language definition and tool developer library (Java and C++) available for download. ACME offers sufficient generality to describe a variety of systems using system structures unlike many other domain-specific ADLs which cannot cope with a wide range of systems. Besides, Armani [Mon98] extends ACME and is a language that is used to express architectural constraints over ACME architectures. However, ACME/Armani does not support dynamism but allows any attribute to be in a property list, i.e., supports non-functional properties (e.g., security properties). The basic elements of ACME are [BJC05]:

- Components (COMPONENT) are composite computational encapsulations that support multiple interfaces known as ports;
- Ports (PORT) are connected to other components’ ports through the use of connectors (CONNECTOR);
- Connectors support roles (ROLE) which can be attached directly to ports;
- Attachments (ATTACHMENTS) specify PORT-ROLE associations;
- Representations (REPRESENTATIONS) are alternative compositions of components utilised to demonstrate that they can have multiple alternative implementations;
- Properties (PROPERTIES) are <NAME, TYPE, VALUE> triples attached to ACME elements as annotations;
- Architectural styles (STYLE) specify types of components, connectors, properties, and rules.

Below, we demonstrate some of the described elements of ACME/Armani in the client-server style that defines two types of components, clientT and serverT, and a tcpT connector which uses a TCP protocol. Besides, this style specifies design rules and limitations to the vocabulary types.

```
STYLE client-server-style = { 
    COMPONENT TYPE clientT = { Port makeCall; }; 
    COMPONENT TYPE serverT = { Port receiveCall; }; 
    DYNAMIC CONNECTOR TYPE tcpT = { 
        ROLES { caller; callee; }; 
        PROPERTY blocking : boolean << default : boolean = true>>; 
        PROPERTY callerAddress : string; 
        PROPERTY calleeAddress : string; 
    FORALL comp in self.component 
        (DeclaresType(comp, clientT) AND SatisfiesType(comp, clientT)) OR 
        (DeclaresType(comp, serverT) AND SatisfiesType(comp, serverT)); 
    FORALL conn in self.connectors DeclaresType(conn, tcpT) OR (DeclaresType(conn, tcpT); 
```


However, ACME/Armani does not currently support constructs for dynamic runtime reconfiguration. Hence, we have extended it with additional features to allow this ADL to be used to describe dynamic reconfigurations. Actually, there were few tries to overcome this issue. Dynamic ACME [Wil01] is one of them, which is an ACME extension for modeling dynamic architectures. However, it is mainly focuses on evolution of specifications but not on support of runtime reconfigurations.

The problem of specifying and analysing dynamic software architectures is also studied in [ADG98] where ADL Wright is extended with dynamic behaviour. In [EHK05] Wright is extended with non-functional properties. Dynamic self-organising distributed component software architectures are described in [Geo02] using ADL Darwin. An up-to-date survey about adaptive software architectures is provided in [Col07] where the concept of ontogenic adaptation is applied to software systems. The author sets out what properties a software system would require for it to be considered ontogenically adaptive in terms of reconfiguration, regulation and management. The author identifies and analyses a number of distinguishing characteristics of adaptive architectures and briefly discusses related areas of research that were not covered. This review concludes by presenting a comparative table with a summary of the characteristics of the reviewed frameworks. However, all the research works described above do not consider security threats as the main driven force of dynamic reconfiguration and adaptation.

### 6.4.2 GIZKA.ADL

As previously mentioned, we need an ADL which is capable to specify dynamic architectures from different domains. Hence, for such purpose we have selected ACME/Armani because it is not domain-specific and sufficient to describe a wide range of systems and it comes with various tools. However, ACME/Armani does not support constructs for security properties as well as dynamic reconfiguration and adaptation. Therefore in this section, we present the part of our language called GIZKA.ADL which is based on ACME/Armani [GMW00,Mon98] and employed for specifying dynamic adaptation and reconfiguration mechanisms for component or service-based systems in order to improve their security. The structure of the whole system is depicted in GIZKA.ADL while security related features of components such as security properties or security interfaces are expressed in GIZKA.ESCL.

**Programmed and runtime reconfigurations**

Our approach can be used for describing both programmed and runtime reconfigurations. Programmed reconfigurations capture changes that can be known at design time (ADL level). For example, ‘IF-ELSE’ conditions at the ADL level are used to express the system’s changes that can be foreseen. However, programmed reconfigurations are not capable to handle...
undefined changes in the system and the system’s environment that may happen in future. We follow the approach proposed by [BJC05] that builds general invariants into the programmed specification of the system and accepts changes at runtime if invariants are not violated. It should be mentioned that runtime reconfigurations depend on the actual implementation of the system. Besides, the runtime reconfiguration should not violate the programmed reconfiguration, i.e., dependencies at design time should not be violated by changes at runtime. Also the dynamic constructs are chosen at runtime. By adding additional constraints at the ADL level it is possible to control integrity of conformance of the programmed and runtime reconfigurations.

The programmed reconfiguration is integrated with the runtime reconfiguration in the following way. Firstly, the programmed reconfiguration is specified in GIZKA and converted to the ADL-level script. Then, this script is applied to manage the system at runtime. However, if changes take place at runtime, the initial script is updated according to new requirements. During the update, the modified script is evaluated in order to verify if it does not violate the ADL-level script (which then updated). This process is called the script modification. We should mention that runtime reconfigurations include the script modification and the actual system reconfigurations.

**ACME/Armani extensions**

ACME/Armani has been enhanced with additional features for depicting dynamic changes of the architecture when security environment changes or a user wants to change security requirements. As previously mentioned, these features were introduced in order to specify and manage programmed and runtime reconfigurations and their security:

- Runtime conditions express when and how reconfigurations should take place together with a specification of what should be changed:
  - **IF** (<conditions>) { <actions> } **ELSE** { <actions> }

For example, if the TCP SYN attack is detected the system should block an attacker first, then increase the size of the connection queue, and decrease the time-out waiting for the three-way handshake. Such fact can be demonstrated as follows:

```
IF (SecurityEvent1.NAME == SAO.Attack.DoSAttack/DDoSAttack.TCPSYNAttack) {
    BLOCK_IP = 192.168.9.34; // attacker’s IP
    INCREASE_CONNECTION_QUEUE = +50;
    DECREASE_TIMEOUT = -1ms;
}
```

In this example, the first line states that if the event (the class ‘Event’ from the security asset-vulnerability ontology) equals ‘TCPSYNAttack’ then several commands should be
executed. The first command ‘BLOCK_IP’ blocks the attacker’s IP address (i.e., 192.168.9.34). The second command ‘INCREASE_CONNECTION_QUEUE’ increases the connection queue by 50 connections. The last command ‘DECREASE_TIMEOUT’ decreases the timeout by 1 millisecond.

- Internal actions are used for specifying connections between components as well as between roles and ports, between connectors and ports, or between ports and functional or security properties.
  - <element> BIND <element>
  - <element> UNBIND <element>
  - <element> REBIND <element>

For example, in a case of the flooding DRDoS attack such as a Fraggle attack against a server (S1) used to register users and consequent server’s unavailability, a client (C1) is allow to connect to another server (S2) that provides same functionalities. This fact can be described as follows:

```plaintext
IF (SecurityEvent1.NAME == SOA.Attack.DoSAttack.DRDoSAttack.FraggleAttack) {
  C1 UNBIND S1;
  C1 BIND S2;
}
```

- External actions describe creating, destroying, or failing of components, connectors and representations. These commands are possible if they are no longer involved in attachments.
  - CREATE <component, connector, representation>
  - DESTROY <component, connector, representation>

For example, the fact that a component C2 should be created while a component C1 is destroyed can be stated as follows:

```plaintext
CREATE C2;
DESTROY C1;
```

- Relations define runtime dependencies among elements used in dynamic reconfigurations. They show that creating and destroying of elements depend on creating and destroying of other elements in order to avoid architectural mismatches during converting the runtime level script to the ADL-level script.
  - RELATIONS { <dependencies> }

162
For example, a component C1 should be destroyed, however, it is known that its connector role C1.Con.Role is connected to the port of another component C2, i.e., to C2.Port. This fact can be demonstrated as follows:

\[
\text{RELATIONS} \{ \text{ATTACHMENTS} \{ C1.\text{Con.Role TO C2.Port} \} \}
\]

- Another extension allows attachments to be specified for both types and instances. The required instance is chosen at runtime according to specified policies.
  - \(<\text{element: element.type}> \text{DYNAMIC} <\text{KEYWORD}> <\text{element: element.type}>\)

The keyword \text{DYNAMIC} placed in front of the keywords described above (\text{BIND}, \text{UNBIND}, \text{REBIND}, \text{CREATE}, \text{DESTROY}) allow to define connections among elements and among their types.

For example, the fact that a client C1 can connect to any instance of a storage server SS (there are three instances of SS: SS1, SS2, and SS3) can be illustrated as follows:

\[
\text{C1 DYNAMIC BIND SS;}
\]

Additional features
GIZKA.ADL also supports other keywords not directly used to specify dynamic reconfigurations:

- According to the Basic Security Theorem, if an initial state is secure and all allowed state transitions are secure as well then a subsequent state is also secure [BL76,Bel88,McL85].

To specify a baseline (also called a basecurve) of the system through the use of current system components, the keyword \text{BASELINE} together with \text{TRESHOLD} and \text{ALERT} are employed. Also it is necessary to mention that additional system components can decrease a baseline, i.e., new components add more possible design flaws and vulnerabilities resulting in a less secure system. Yet, connected and disconnected components or clients decrease a baseline too. Moreover, an ideal baseline of a secure system is when all external connections are forbidden, all inner components are trusted, and all inner connections are secured. Hence, the greatest weakness of the system is its external connections.

Figure 6.9 illustrates a baseline and a threshold which shows that extreme countermeasures such as system alerts for system administrators and reconfiguration are performed only if security goes below a threshold. The system can do nothing or just alert a system administrator if the current level of security is between a threshold and a baseline. Also, different baseline evaluation mechanisms for different types of systems can be employed.
Figure 6.9. Baseline

For example, for P2P systems the speed of disappearance of peer neighbours can be used as a baseline. Another method to develop a baseline is to make system snapshots and then compare various parameters. In addition, connections among components can be evaluated in the following way: 1) The system checks if encryption is used or not, 2) Then, the system compares security algorithms and, based on the results, checks if system’s security is above or below a baseline.

- The keyword **WRAP** is employed to wrap a non-secure element with additional security countermeasures in order to create a secure element. For example, the fact that a non-secured port called ‘reqP’ is wrapped by using the AES or DES algorithms is specified in the following pseudocode:

```
PORT reqP;
PORT reqP.Secure {
    SecurityAlgorithm.AES ALIAS A;
    SecurityAlgorithm.DES ALIAS B;
    WRAP (reqP, A OR B) = WRAP -> check if A is supported on both sides and WRAP -> return | check if B is supported on both sides and WRAP -> return | NO WRAP -> return;
}
```

The pseudocode inside the brackets verifies if both sides support AES and then wraps the port ‘reqP’, otherwise it checks if DES is supported and then wraps the port. If none of the algorithms is supported then the port is left untouched.

- Required and ensured security properties expressed in GIZKA.ESCL can also be specified using the keyword **SECPROPERTIES** {<security properties>} (in addition to the representation described in the previous sections).

**An example**
In this section, we present an example of the mobile virtual office system introduced in Section 6.2 and illustrate some of the concepts described earlier. In this example, for securely exchanging text messages, users can switch from SSL to IPSec that supports compression of data if network bandwidth is low. If intrusion detection components cannot resist Smurf attacks on the Message Storage 1 (MS1) then it simply dies and clients (Client1 (C1) and Client2 (C2)) connect to Message Storage 2 (MS2). We describe a simplified example, as illustrated on Figures 6.1-6.2 and Figures 6.10-6.11, using GIZKA for depicting dynamic changes of the architecture when security environment changes or a user wants to change security requirements. Security related features of peers are expressed in GIZKA.ESCL. All cryptographic keys have been securely distributed by a key distribution component (KDC) (not shown in Figures 6.10-6.11).

Firstly, we introduce the client-server style where we define two types of components, clientT and serverT, and a tcpT connector which uses a TCP protocol, as shown in Figure 6.12. Then, we define design rules for this style and limit the vocabulary types used in this style to clients and servers.

```gizka
class clientT {
  port makeCall;
}
class serverT {
  port receiveCall;
}
class tcpT {
  role caller, callee;
  property blocking : BOOLEAN = true;
  property callerAddress : STRING;
  property calleeAddress : STRING;
}
```

**Figure 6.10. The part of the system before Smurf attacks**

**Figure 6.11. The part of the system after Smurf attacks**

**Figure 6.12. The client-server style**
Then we can extend this example style to create a substyle that introduces a new subtype of servers called a message storage server and a key distribution server which also use a TCP protocol, as illustrated in Figure 6.13.

```
STYLE system-cs-style EXTENDED WITH client-server-style {
  COMPONENT Type msServerT EXTENDED WITH serverT {
    PORT receiveCall = {
      PROPERTY query-language : protocol << default : protocol = TCP>; }
    PROPERTY primary-server : BOOLEAN << default : BOOLEAN = FALSE >>;
    DESIGN INVARIANT tcp-callee.query-language == TCP;
  }
  COMPONENT Type kdServerT EXTENDED WITH serverT {
    PORT receiveCall = {
      PROPERTY query-language : protocol << default : protocol = TCP>; }
    PROPERTY primary-server : BOOLEAN << default : BOOLEAN = FALSE >>;
    DESIGN INVARIANT tcp-callee.query-language == TCP;
  }
}
```

Figure 6.13. The system-cs substyle

Now we describe the example system that uses the system-cs-style style, as shown in Figure 6.14, where message servers, MS1 and MS2, support message exchange functionality (MsgExchange(MSG)). We consider that all cryptographic keys have been securely distributed by a KDC (not shown here). C1 signs, encrypts and securely sends messages to C2 through MS1 or through MS2 if there is a Smurf attack against MS1 or network bandwidth is low. The connectors between C1, C2 and MS1 and MS2 are declared and hooked together. However, connections between components can be created only if security properties are satisfied.

```
SYSTEM example-system : system-cs-style = {
  COMPONENT MS1 : msServerT = new msServerT {
    PORT adminPort;
    PROPERTY primary-server : BOOLEAN = FALSE;
    SECPROPERTIES {
      Protected(MSG,C1,C2).
      InternalTransfer(MSG,C1,C2)←Signed(MSG,K[-C1]),TrustedPath(C1,C2).
      TrustedChannel(C1,C2,SSL). }
  }
  COMPONENT MS2 : msServerT = new msServerT {
    PORT adminPort;
    PROPERTY primary-server : BOOLEAN = FALSE;
    SECPROPERTIES {
      Protected(MSG,C1,C2).
      InternalTransfer(MSG,C1,C2)←Signed(MSG,K[-C1]),TrustedPath(C1,C2).
      {TrustedChannel(C1,C2,SSL) OR TrustedChannel(C1,C2,IPSec),
      Time(<9am,5pm>),Probability([0.9,1])}. }
  }
  COMPONENT C1 : clientT = new clientT {
    SECPROPERTIES {
      Signed(MSG,K[-C1]).
      PairOf(K[-C1],K).
    }
  }
}
```
The ensured property of MS1 InternalTransfer states that the message is transferred to C2 if it is signed by C1 and a trusted path is established. A trusted channel is established between C1 and C2 using SSL or IPSec security algorithms. The difference between security properties of MS2 and MS1 is that MS2 provides IPSec support from 9am to 5pm with probability interval
[0.9,1](90%-100%) because it is known through the use of traffic patterns that the network may be overloaded from 9am till 5pm. The security properties of $C_1$ ensure that a message is distributed from $C_1$ to $C_2$ if the key is digitally signed by $C_1$ and a trusted path is established between $C_1$ and $C_2$ in order to provide integrity and confidentiality of the key. The security properties of $C_2$ are similar to $C_1$.

As previously mentioned, runtime reconfiguration depends on the actual implementation of the system. While programmed reconfigurations capture changes known at design time, they are not capable of handling undefined and future changes in the system and the system’s environment. Moreover, programmed reconfigurations should not be violated by runtime reconfigurations, i.e., changes at runtime should not violate dependencies at design time (specified using the keyword ‘RELATIONS’ and ‘DYNAMIC’).

6.5 Security patterns

As a countermeasure against DDoS flooding attacks we propose the structural reconfiguration and adaptation. Hence, to express dynamic reconfigurations in a faster and easier way, in this section we introduce security patterns (SecPs). They are integrated into the reference architecture, i.e., SECROBAT. Besides, they express scenarios about what and how components and the system should do in various cases of security attacks or other environment changes. The main advantage of using SecPs is that many of them combine the knowledge of security experts about how to solve certain security issues. If we simply use already developed SecPs, however, all SecPs are specified in a human language (e.g., English) that is not understandable by machines or software agents. They can be stored as a collection of patterns and instantiated and redefined at design time as well as at runtime. For example, a system identifies a security attack and emits an alert sent to a system administrator. This person may choose a proper defence from a collection of countermeasures defined using SecPs, and consequently, enforce the deployment of these SecPs.

We collect SecPs from various sources [YB97,SNL05,SP07], modify them for our needs, and create two additional patterns including ‘PureP2PPattern’ and ‘SuperP2PPattern’. Furthermore, we define the security pattern ontology (SPO) used as a vocabulary (similar to other our security ontologies) to share information about SecPs and express dynamic reconfigurations in a way comprehensible to humans and software agents as well. SecPs are expressed in GIZKA, as demonstrated below. SPO is closely related to and uses the security attack ontology (SAO) and the security defence ontology (SDO) described in the previous chapter.
6.5.1 Definition

Security patterns (SecPs) are defined in [SNL05] in the following way: “Security patterns are an abstraction of business problems that address a variety of security requirements and provide a solution to the problem. They can be architectural patterns that depict how a security problem can be resolved architecturally (or conceptually), or they can be defensive design strategies upon which quality security protection code can later be built.” A more generic definition of SecPs is extracted from [SP07] as follows: “A security pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution.” We follow the first definition because it is more specific.

6.5.2 Security pattern template

Our SecPs can be represented using a general template [SNL05] modified for our needs that includes:

- Name – every SecP should have a name that becomes a part of SPO;
- Comment – description of the addressed security issues;
- Forces – used to describe the motivations, constraints and requirements that affect the security problem;
- Solution – a brief description be the approach and associated mechanisms given in details;
- Strategies – define different ways a SecP may be implemented and deployed;
- Consequences – describe the results and trade-offs of using a SecP as a countermeasure;
- Security factors and risks – introduce factors and risks to be considered if a SecP is applied;
- Reality checks – used to describe a set of review items to identify the feasibility and practical reasons for using the pattern;
- Related patterns – list other related patterns.

In addition, we consider the elements such as EXAMPLES and STANDARDS. While the former are used to provide cases where the particular pattern can be employed the latter specify various security standards and mechanisms utilised by the pattern.

Every SecP starts with the keyword SECPATTERN and all information related to SecPs is placed into security pattern templates. The template can be specified as illustrated in Figure 6.15.

```
```
6.5.3 Security pattern ontology

In this section, we present the Security Pattern Ontology (SPO) and specify it using GIZKA. We develop two SecPs for the message layer (also called the sniffing layer) (the class ‘MessageLayer’) and four for the system layer (the class ‘SystemLayer’), as illustrated in Figure 6.16.

‘SystemLayer’ enables secure client-to-client (or P2P), client-to-server, and server-to-server communications and specifies P2P reconfiguration patterns. ‘MessageLayer’ represents mechanisms needed to secure messages among components or services. We introduce layers described above and SecPs and specify some of them using GIZKA pseudocode. It is necessary to mention that SecPs can be described in various ways that depend on the realisation of the system. Our pseudocode just shows the structure and logic of our SecPs. The keyword ENFORCE is used to demonstrate that in the particular case the certain SecP can be employed. The selection can be performed by the manager of the system (e.g., a system administrator) or the software agent. Other examples of utilising SecPs can be found in Chapter 7 and [SNL05] (with a lot of examples, best practices, and reality checks). It should be mentioned that security patterns specified below can be instantiated and redefined later at the stage of defining an architectural style and at runtime.

System layer

SecPs for the system layer are represented by the classes ‘PureP2PPattern’, ‘SuperP2PPattern’, ‘SingleAccessCheckPoint’, and ‘SecureCommunication’, as shown in Figure 6.17.
As previously mentioned, one of the strategies to protect our system from security attacks is to reconfigure it from the super P2P architecture (during normal regime) to the pure P2P architecture (during security attacks). Hence, we define ‘PureP2PPattern’ and ‘SuperP2PPattern’. The detailed description of both architectures has been presented in Chapter 3. ‘PureP2PPattern’ specifies the system’s strategies during security attacks. More specifically, it enforces components to connect certain components or types of components, as shown in Figure 6.18.

(PureP2PPattern)

Figure 6.18. PureP2PPattern

‘SuperP2PPattern’ (illustrated in Figure 6.19) represents the super P2P architecture used during normal operation of the system. It is used together with ‘PureP2PPattern’ to allow the system to survive during security attacks.

(SuperP2PPattern)
//A strategy that specifies what type of connection is allowed.
STRATEGY AllowedConnections() {/* can be defined later*/
//Components can connect only components of the same type
  Comp DYNAMIC BIND ENTITY.SAME;
}  
STRATEGY AllowedSameP2PConnection(ENTITY Comp) {
  //Components can connect only components of the same type
  Comp DYNAMIC BIND Comp2.TYPE;
}  
STRATEGY AllowedTypeSuperP2PConnection(ENTITY Comp, ENTITY Comp2) {
  //Components can connect only specially specified components
  Comp DYNAMIC BIND SuperComp;
}  
STRATEGY AllowedSuperP2PConnection(ENTITY Comp, ENTITY SuperComp) {
  //Components can connect only through super peer
  Comp DYNAMIC BIND SuperComp;
}

Figure 6.19. SuperP2PPattern

(Example)

For example, if there is a Smurf attack against the mobile virtual office system, the system needs to reconfigure itself to resist this attack (as mentioned in Chapter 5, there is no full protection against such attacks if traditional security mechanisms are used) and change its structure from super P2P to pure P2P. This case can be expressed in the following way:

IF(SecurityEvent1.NAME == SAO.Attack.DoSAttack.DRDoSAttack.SmurfAttack) {
  ENFORCE SPO.SystemLayer.PureP2PPattern.AllowedAnyP2PConnection(Components);
} ELSE {
  ENFORCE SPO.SystemLayer.SuperP2PPattern.AllowedSuperP2PConnection(Components);
}

In this example, the system compares the emitted security event (‘SecurityEvent1’) with the specification of a Smurf attack. If the attack is detected, then a P2P pattern is enforced, otherwise, Super P2P pattern is deployed.

(SingleAccessCheckPoint)

The class ‘SingleAccessCheckPoint’ enforces a single checkpoint of entry to the services and peers that provide a login prompt. In the Web context, it is usually implemented by forms-based authentication and SSL. The process starts from identification when a client enters a login. Then, authentication happens when a client enters a password or a secret phrase. After this, the process of authorisation occurs that allows a client to perform certain tasks. Other processes are auditing and accountability. The class ‘SingleAccessCheckPoint’ is illustrated in Appendix C.

(SecureCommunication)

The class ‘SecureCommunication’ [YB97] describes securing transport and network layers for client-to-server, client-to-client (P2P), and server-to-server communications through the use of
the HTTPS, SSL, TLS, and IPSec protocols. It applies the strategy of encrypting traffic using specified algorithms. The class ‘SecureCommunication’ is demonstrated in Appendix C.

(Example)
For example, in the mobile virtual office system, the case that the manager wants to secure the connections between two components such as C1 and C2 using the SSL protocol. It can be specified in GIZKA in the following way:

```
ENFORCE SPO.SystemLayer.SecureCommunication(C1,C2,SSL);
```

Message layer
Attackers may try to sniff the connections among components, hence, there is the need to secure such links. One of the best solutions is to protect, inspect, and verify messages used by components to communicate to each other. In our approach, the message layer (used to mitigate sniffing attacks) is responsible for those tasks. SecPs for the message layer are represented by two classes ‘MessageInspector’ and ‘MessageInterceptorGateway’ (we borrowed both names of classes from [SNL05]), as shown in Figure 6.20.

![Figure 6.20. MessageLayer](image)

(MessageInspector)
This class inspects messages on all levels of the OSI models. For example, ‘MessageInspector’ can be employed to check application layer messages such as SOAP messages. Also, this pattern employs various security standards including XMLEnc, XMLDSig, SAML, XKMS, AES, and 3DES. The SecP that is closely related to ‘MessageInspector’ is ‘MessageInterceptorGateway’ introduced in the next section. ‘MessageInspector’ works in the following way. Firstly, it checks if there are any anomalies in the message. Then, if there are security related issues then this pattern sends an alert to the system’s manager. But if the message is clean then it is forwarded to its destination. The illustration of this pattern in GIZKA pseudo code is given in Appendix C.

(MessageInterceptorGateway)
This security pattern is employed as a single entry point in order to provide central security enforcement for all incoming and outgoing messages. Besides, it creates, modifies, and administers the system’s message security policies. Also, it uses various security standards and mechanisms including XMLDSig, XMLEnc, SAML, WS-* , AES, and 3DES. ‘MessageInterceptorGateway’ checks messages using ‘MessageInspector’ and enforces security policies, as shown in Appendix C.

(Example)

In this section, we just demonstrate a short example of how SecPs can be used. For example, if there is the mobile virtual office system (introduced in the previous sections). One of the IDC detects a sniffing attack (explained in Chapter 5) and reports to the manager about the case (described in Chapter 5). Then, this manager can decide to enforce ‘MessageInspector’ pattern. This scenario can be specified in the following way:

```plaintext
IF(SecurityEvent1.NAME==SAO.Attack.SniffingAttack) { ENFORCE SPO.MessageLayer.MessageInspector(Message); }
```

As previously described to indicate the enforcement of SecPs, we use the keyword ENFORCE. This keyword shows that the particular pattern should be used in the particular case. The selection and deployment can be done by the human who can be the manager or by the software agent.

Relationships among various security patterns

Various security patterns can be organised in more complex patterns. For example, one of the complex patterns is illustrated in Figure 6.21. ‘SuperP2PPattern’ uses ‘SingleAccessCheckPoint’.

![Figure 6.21. Relationships among security patterns](image)

As previously mentioned the actual specification and deployment of the patterns depends on the technologies and programming languages used to implement them. In our case, they are defined in (re)configuration scripts used to manage the actual system.
6.6 Summary

One of the main objectives of our research work is to develop an approach to achieving higher level security for component (or service) based software systems. Hence, we have created the reference architecture called SECROBAT used to develop component based software systems that are secure and attack protected and can detect and mitigate security attacks and other environment changes, as presented in Chapter 3. Also, SECROBAT’s components need to communicate with each other because of their distributed nature. They require a vocabulary to share information regarding various security aspects such as security attacks and defenses. Hence, we have created security ontologies as the common basis for the components to communicate each other, as described in Chapter 4 and Chapter 5.

Furthermore, to reduce the cost of developing and managing of the systems based on our approach we may need to create a framework that unites different parts of our approach. We need a way to describe them in a formal manner in order to allow component (service) based software systems to be secure, robust, and easily manageable. Hence in this chapter, we have demonstrated the language called GIZKA utilised to specify dynamic software systems, their dynamic programmed and runtime reconfigurations and adaptations, their security properties. This language together with security ontologies is also employed to define security attacks and defenses in order to detect them, correlate and anti-correlate them, and develop and deploy countermeasures using security patterns. Moreover, we have developed the security pattern ontology and reasoning rules for components and services to improve security and flexibility of GIZKA.
Chapter 7

A social network system as a case study

In the previous chapters, we have described the need for our approach and introduced all its parts: (1) the reference architecture called SECROBAT with defensive components and the hybrid pure P2P/super P2P architecture, (2) security ontologies including the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), the security function ontology (SFO), and the security attack and defense ontologies (SAO and SDO), (3) the language for managing security called GIZKA which has several subsets including GIZKA.ESCL, GIZKA.ATTACK and GIZKA.ADL. GIZKA.ATTACK allows to describe security attacks and defenses in order to identify security attacks, analyse and correlate them, and then anti-correlate (i.e., choose a proper countermeasure) them with proper countermeasures using GIZKA.ADL. GIZKA.ESCL together with GIZKA.ADL allow to specify security properties of component or service based software systems and define how they can dynamically adapt and reconfigure their architecture because of environment or user requirement changes. In this chapter, we demonstrate our approach through the use of the example of the social network system. We organise our case study as a business plan with of the necessary technical details to demonstrate the real-world applicability of our approach.

7.1 Business plan

In this section, we present a business plan of the social network system (SNS) based on our approach in order to show that it can be adopted for the real world. Besides due to an artificial nature of this business plan, it does not cover such important issues as a brief history of the company, the company’s name, the business address, contacts, the management team’s background and profile, and the financial profile including last year’s annual revenues (sales turnover), 5-year forecast on sales and capital investment needs, assumptions behind forecast, the current ownership structure (owner’s equity, angels, others), and future plans.

7.1.1 Social network system

In this section, we describe our SNS and its specific benefits. Besides, we introduce the implementation strategy and report the degree of current development including a brief description of tasks performed.
Operational use and benefits of our SNS

Our social network system (SNS) is based on our approach which is used for implementing various types of distributed software systems including instant messaging systems, audio/video conferencing, collaborating systems, social networks, online video, and online games, etc. The SNS can be easily created and extended because our approach employs several technologies and methodologies including a component-based approach, various peer-to-peer (P2P) based technologies, Web services, Semantic Web etc. At the same time, our SNS can be highly secure and robust because it can adapt and reconfigure its structure to identify, analyse, and withstand various security attacks especially Distributed Denial of Service (DDoS) and Distributed Reflective Denial of Service (DRDoS) attacks (there is no 100% defense against them). Moreover, the SNS can detect and resist against many other security attacks such multi-phased distributed attacks using defensive components. Also, the system can be healed and repaired and it is quite difficult to block it even if an attacker tries to cut off some key parts of the system’s network. The system finds a way to reconfigure and continue operating. Since the SNS has been implemented in Java [Java06] and Groovy [Groovy07], it can be deployed on any host where the Java Virtual Machine (JVM) is installed (e.g., it can be launched in a web browser as a Java applet). Finally, since we use the concept of open source the SNS is going to be cheap to develop and deploy.

Status

We have developed the reference architecture SECROBAT with the key features including defensive components (intrusion detection components, honey-pot components, and key distribution components) and the adaptive and reconfigurable hybrid pure P2P/super peer structure. Moreover, we have created a language for managing security called GIZKA based upon security ontologies (specified in OWL [OWL07] using Protégé [Protege07]) that allows depicting dynamic changes of the architecture when security environment changes or if a user wants to change security requirements. Furthermore, we have implemented a prototype which is based on such technologies as Java, Groovy, P2P, etc.

7.1.2 Market

In this section, we describe the size of the market, its growth potential, demand opportunity and customer preferences. Besides, we explain how we intend to reach our market including our strategy in terms of pricing, promotion, selling and distribution.

Target market

Since we push the social network system (SNS) then primarily at this stage we focus on the social networking market. After implementing the SNS, we will start competing with the current
major players at the market of social networks, i.e., Youtube [Youtube07], MySpace [MySpace07], Facebook [Facebook07]), and their clones providing resembling services, and various P2P systems, instant messengers specializing in providing different services (video/or music sharing, blogs, online TV, online games, audio/video conferences, etc.). According to Nielsen/NetRating report (global leader in Internet media and market research) [BH06] that announced that April’s (2006) top 10 social networking sites collectively grew 47% year over year, increasing from an unduplicated unique audience of 46.8 million last year to 68.8 million in April 2006, reaching 45% of active Web users. Owing the fact that our SNS may be capable to provide various kinds of services including social networks, online TV and video, online games, instant messaging, collaborative works, etc., we consider to focus primarily on such niches. Moreover, we feel that our SNS is able to provide better functionality, cheaper support, and, what is more important, better trust, security, and privacy.

Marketing plan
First, we implement the social network system which allows users to chat, watch video online, publish their materials, play online games together, and etc.

The major revenue will be earned from advertisements. To prove that our reflections are not unfounded, we provide some figures (on the sole U.S. market): U.S. marketers spent $350 million in 2006, an increase of 25% over previous estimates and the estimates for 2010 are also up 16% to $2.2 billion, with the social network ad spending projected to account for 8.5% of the U.S. ad market by 2010 [WE07]. The inclusive advantage of targeting ads market is that today the population of users (of such networks) is geographically wide (especially!) and numerically large. Apart from that, the online gaming market is also under our steadfast attention and it is expected to grow enormously and reach US$1.3-2 billion in 2009 [Nys05]. Thus, we tie our expectations with different segments of the game industry (distribution services, casual Web-based services, in-game advertising, and commodity exchange).

At the beginning, the price will be negotiated individually. Later, four types of packages for advertisements will be created: ordinary, silver, gold, and platinum. Subscriptions for ordinary users will be free. However, if users want additional support they need to pay monthly charge fees for such maintenance.

After capturing the market of social networks, we are planning to implement a search engine that further can collaborate with the similar powerful search and advertising products issued by Google [Google07] and Microsoft [Microsoft07].

7.1.3 Industry analysis
In this section, we introduce what challenges we see in terms of the SNS, the market and the industry we are entering.
Challenges

Once the SNS is completely developed and successfully tested, we will definitely tackle with such obstacles as choosing the competent strategy for the promotion of our product and advertising as well. Then, we should solve such problem as building proper relationships among stakeholders. Besides, one of the main concerns regarding our SNS is whether our system is considered to be reliable and secure. One of the primary concerns that might arise will be the privacy issues. Further, in terms of the target market (in our case, it is the market of social networks and online games), we should not leave the concern about such challenge as seeking the balance between membership (of attracting the potential subscribers/or business audience) and monetisation, i.e., in other words how to find the necessary balance between the demographics and expendable incomes for the product. Moreover, problems associated with choosing the competent market strategy and finding and taking our own market niche together with building the database of clients are deserved to be mentioned. Finally, we argue that the SNS may require some alterations or versions in terms of geographical spread of the software product.

7.2 Requirements

In the previous section, we have introduced the business plan for the social network system (SNS) in order to illustrate its applicability to the real world. In this and next sections, we describe the basic consumer and the provider (system) functional and non-functional requirements such as security requirements for the SNS. It is necessary to mention that a provider is a company (also called owner) which implemented its own SNS using our approach. In the context of this thesis, a provider is the same company for which we have presented a business plan in the previous section. A consumer (client) is an actual user of the SNS.

7.2.1 User’s requirements

Functional requirements

In general, a user of our SNS requires different functionalities including chatting, exchanging of instant messages, sharing audio, video, or text files, organising audio and video conferences, watching online TV, working collaboratively with other users, playing online games, sending emails, storing and archiving their data, and so on. Hence, the system should support these functionalities and additionally have a user-friendly interface.

Security and quality requirements

However, the users may be concerned about various issues related to security, availability, and privacy (e.g., the protection of their private data). Besides, security should be transparent for
users because they just want securely utilise services provided by the system without bothering about underlying implementations. Yet network connections between clients and the system should be protected and clients’ data stored on the system storage servers should be encrypted as well while users do not care how security features are realised. Also, the users require that the SNS should be available all the time and be easy to use. Hence, our SNS should support all these requirements.

7.2.2 Provider’s requirements

Functional requirements
From the provider’s perspective, the SNS should meet previously mentioned user’s requirements. Besides, the system should allow to store system and users’ data, organise connections between different parties, and be easily managed. However, these functionalities should be delivered using as least hardware resources and IT staff as possible because of the financial considerations.

Security and quality requirements
Because of the same reason mentioned above, the system and its security should be easily managed and be available all the time as well. Furthermore, security should be transparent for users because they just want securely utilise services provided by the system without bothering about underlying logic and technologies. Network connections between users and the system, and users’ data stored on the system storage servers should be protected. Hence, there should be mechanisms that allow the system to secure network connections and protect users’ data. Moreover, the system should be protected from various computer pathogens including computer viruses, Trojan horses, malware, and spyware. Specifically, the system should identify, correlate, anti-correlate, and mitigate security attacks, especially distributed multi-phased attacks such as Mitnick attacks (treated as a baseline of the minimal level of security), flooding distributed denial of service (DDoS) or distributed reflective denial of service (DRDoS) attacks (e.g., Smurf or Fraggle attacks). The system should also allow to deploy and enforce different defenses and be capable to survive during attacks. Finally, connections between different parties should be encrypted, and organised and governed using security properties of these parties.

7.3 SNS and specification of its architecture
In this section, we describe the SNS and specify its architecture using GIZKA. We define a system architecture design. We demonstrate how initial connections between system’s components are created and governed using security properties of components specified in GIZKA.ESCL. Besides, we illustrate how the system withstands flooding DRDoS attacks such
as Smurf or Fraggle attacks through the use of its dynamic structural reconfiguration and adaptation.

### 7.3.1 Overview and scenarios

In this section, we introduce our SNS and present in more details functionalities delivered by it. This SNS meets user’s and provider’s requirements, as mentioned above. Currently, the SNS (similar to MySpace [MySpace07] or Facebook [Facebook07]) allows users to chat and exchange instant messages securely (like in ICQ [ICQ07]), share various files including audio, video (as in Youtube [Youtube07] and P2P file sharing systems such as eMule [eMule07] or BitTorrent [BitTorrent07]), and text files. Besides, it allows to organise audio/video conferences, do collaborative drawing and collaborative work, play online games, send emails, and store data locally or remotely on hard drives or in databases (e.g., MySQL [MySQL07]), archive user’s data using RAR or ZIP algorithms [WIKI07], encrypt data using various cryptographic algorithms, and so on.

To protect the system (SNS) and users (clients) from various security threats such as computer pathogens or security attacks, we adopt the SECROBAT approach. The SECROBAT-based SNS in the normal regime, as illustrated in Figure 7.1, looks like a super-peer system. All connections between users (Client1, Client2, Client3, Client4, and Client5) and the system (storage servers and defensive components including DC1 and DC2) are controlled by intrusion detection components (IDC1 and IDC2) that serve as proxies for them. Besides, DC1 includes IDC1, a key distribution component (KDC1) and a honeynet component (HC1) while DC2 consists of IDC2, KDC2, and HC2. KDC1 is responsible for distributing cryptographic keys and digital certificates in order to maintain security contracts that govern secure connections between different parts of the system. HCs are adopted to collect new data about security attacks and share it with IDCs. Besides, they study attacks and attract intruders by exposing well-known vulnerabilities. Storage servers (SS1, SS2 and SS3) are responsible for storing personal users’ data, system security properties and policies, etc. Three storage servers are employed to make the data kept on them available all the time. The data is mirrored and each storage server has a synchronised copy. Also, they are directly connected to each other, however, they are linked to clients through IDCs. The main task of the manager (Manager1) is to govern the system and its constituent components. The manager can be a software agent or a human (e.g., a system administrator). A reserved (Reserved1) peer is used if the system needs additional resources in case of security attacks such Smurf or Fraggle attacks.

### Normal regime
If a client (Client1), as shown in Figure 7.1, wants to use this SNS for the first time, she downloads a SNS client program from the website sns.net. After installing it, she specifies her requirements that she wants to chat and share files with clients from Oceania in their 20th from 9am till 5pm. A system manager (Manager1) checks if Client1 has enough funds to use the system and decides what security policies should be enforced for Client1, and what security techniques (SSL and 3DES) should be used because security measures may reduce performance or network bandwidth. Manager1 also stores its data on a storage server (SS1). In addition to chatting, clients can do other collaborative activities including conducting videoconferences.

Dealing with security attacks
Our SNS also has to deal with many security threats including various security attacks such as multi-phased distributed attacks (e.g., Mitnick attacks) and DDoS and DRDoS attacks (e.g., Smurf or Fraggle attacks). Hence in this section, we present several possible attack scenarios.

Scenario 1
Attacker, as illustrated in Figure 7.2 can try to intercept a connection between DC2 and Client4 in order to get Client4’s private information. Hence, this connection has to be secured. It can be done if the network links between the system and clients are encrypted and inspected by IDC2. Also, it is difficult for Attacker to hack communications between Client1 and other peers because IDC1 and IDC2 detect cases of intrusion, study log files, and use information about the studied attacks from HC1 and HC2. Messages are also encrypted using cryptographic keys generated and distributed by KDC1 and KDC2.
Scenario 2

Also, storage servers as well as some other SNS components (e.g., DCs) can be attacked by using different types of attacks including sniffing, various brute-force flooding DDoS/DRDoS attacks which deplete network bandwidth (Smurf or Fraggle), or multi-phased distributed attacks such as Mitnick attacks. It can be a major problem for the provider. Besides, it can be not economically feasible only to add additional hardware servers because there are possibilities that they will not be needed in future. Hence, there should be other approaches to overcome this issue. One of them is to allow the gaming system to reconfigure dynamically in order to resist attacks and distribute load among storage components.

In addition, DCs are used to protect the SNS. More specifically, as illustrated in Figure 7.2, when Client1 is under attack, IDC1 tries to protect it and forwards malicious traffic to HC1 that studies and traces-back to Attacker. To withstand flooding DDoS or DRDoS attacks, the system changes its structure, as shown in Figure 7.2. The additional IDC3 is added using Reserved1 in order to resist attacks and protect system components and clients. Also, every DC looks after each other. All connections among system components and clients are encrypted. Clients can connect to other clients directly. For example, Client1 and Client2 can still chat even during attacks. Also, if the SNS provider needs more resources, the system can try to distribute load dynamically and evenly. To minimize expenses, the SNS provider uses Manager1 to operate and control the system and utilise Client3 and Client4 as storage servers, however, system data stored on these clients should be encrypted.
In the following sections, we specify our SNS and describe how it can deal with security attacks described above.

### 7.3.2 Style PeerStyle

In this section, we define the style ‘PeerStyle’, as illustrated in Figure 7.3, which represents the behaviour of peers (components) and the whole system. We specify types of components including IDCs, KDCs, HCs, Managers, SSs, and Reserved. Also, we define ordinary and secure ports that adopt the 3DES encryption algorithm. Finally, we specify what types of software components are allowed to connect to other types.

```plaintext
STYLE PeerStyle = {
    SASO.SecurityAlgorithm.SAlgEncryption.SAlgSymmetric.3DES ALIAS 3DES;

    COMPONENT TYPE IDCT = {
        //insecure simple ports
        PORT reqP;
        PORT proP;
        //secure ports are created using wrappers for simple ports
        SPO.SystemLayer.SecureCommunication {
            PORT reqP.Secure { WRAP (reqP, 3DES); }
            PORT proP.Secure { WRAP (proP, 3DES); }
        }
    }
    COMPONENT TYPE KDCT = {…}
    COMPONENT TYPE HCT = {…}
    COMPONENT TYPE ManagerT = {…}
    COMPONENT TYPE SST = {…}
    COMPONENT TYPE ReservedT = {…}
    COMPONENT TYPE ClientT = {…}

    ATTACHMENTS {
        SPO.SystemLayer.P2PPattern.AllowedConnections {
            ClientT DYNAMIC BIND ClientT OR IDCT;
        }
        SPO.SystemLayer.SupersetP2PPattern.AllowedConnections {
            IDCT DYNAMIC BIND IDCT OR KDCT OR HCT OR ManagerT OR SST OR ReservedT;
            KDCT DYNAMIC BIND IDCT OR KDCT;
            HCT DYNAMIC BIND IDCT OR KDCT;
            ManagerT DYNAMIC BIND IDCT OR KDCT OR HCT OR SST OR ReservedT;
            SST DYNAMIC BIND IDCT OR SST;
            ReservedT DYNAMIC BIND IDCT OR ReservedT;
            ClientT DYNAMIC BIND IDCT;
        }
    }
}
```

Figure 7.3. The style ‘PeerStyle’

As shown above, in this style we define that every system component has secure and insecure ports. Besides, we specify several rules to restrict connections among various components on the ADL-level:

- IDCs can connect any types of components (P2PPattern);
HCs can connect only IDCs directly and other HCs through the use of IDCs (SuperP2PPattern);  
KDCs can connect IDCs directly and other KDCs only using IDCs as proxies (SuperP2PPattern);  
The manager (Manager) can connect to any system component directly, however, it can connect to other managers through IDCs (SuperP2PPattern);  
SSs can connect only IDCs or other SSs;  
Reserved components can connect only other reserved components and IDCs;  
Users (clients) can connect only other users (if they are allowed) and IDCs only. Users communicate with other system components through IDCs that act as proxies.

7.3.3 Specification of the SNS

Now, we demonstrate how to instantiate the system, as illustrated in Figure 7.4. We define only some of the architectural elements here while the rest including the specifications of security properties, security attacks and countermeasures such as reconfigurations to deal with attacks can be found in the next sections.

```java
SYSTEM SNS : PeerStyle = {
    //Instantiation storage servers
    COMPONENT SS1 : PeerStyle.SST = new PeerStyle.SST {
        INTERFACE SS1 {…} …};
    COMPONENT SS2 : PeerStyle.SST = new PeerStyle.SST {
        INTERFACE SS2 {…} …};
    COMPONENT SS3 : PeerStyle.SST = new PeerStyle.SST {
        INTERFACE SS3 {…} …};
    //Instantiation of the manager
    COMPONENT Manager1 : PeerStyle.ManagerT = new PeerStyle.ManagerT {
        INTERFACE Manager1 {…} …};
    //Instantiation of KDCs
    COMPONENT KDC1 : PeerStyle.KDCT = new PeerStyle.KDCT {
        INTERFACE KDC1 {…} …};
    COMPONENT KDC2 : PeerStyle.KDCT = new PeerStyle.KDCT {
        INTERFACE KDC2 {…} …};
    //Instantiation IDCs
    COMPONENT IDC1 : PeerStyle.IDCT = new PeerStyle.IDCT {
        INTERFACE IDC1 {…} …};
    COMPONENT IDC2 : PeerStyle.IDCT = new PeerStyle.IDCT {
        INTERFACE IDC2 {…} …];
```
// Instantiation of HCs
COMPONENT HC1 : PeerStyle.HCT = new PeerStyle.HCT {
  INTERFACE HC1 {…}
};
COMPONENT HC2 : PeerStyle.HCT = new PeerStyle.HCT {
  INTERFACE HC2 {…}
};

// Instantiation of reserved servers
COMPONENT Reserved1 : PeerStyle.ReservedT = new PeerStyle.ReservedT {
  INTERFACE Reserved1 {…}
};

// Instantiation of the clients
COMPONENT Client1 : PeerStyle.ClientT = new PeerStyle.ClientT {
  INTERFACE Client1 {…}
};
COMPONENT Client2 : PeerStyle.ClientT = new PeerStyle.ClientT {
  INTERFACE Client2 {…}
};
COMPONENT Client3 : PeerStyle.ClientT = new PeerStyle.ClientT {
  INTERFACE Client3 {…}
};
COMPONENT Client4 : PeerStyle.ClientT = new PeerStyle.ClientT {
  INTERFACE Client4 {…}
};
COMPONENT Client5 : PeerStyle.ClientT = new PeerStyle.ClientT {
  INTERFACE Client5 {…}
};

ATTACHMENTS {
  // specific storage servers’ attachments
  SPO.SystemLayer.SuperP2PPattern.AllowedConnections {
    SS1 DYNAMIC BIND SS2;
    SS2 DYNAMIC BIND SS3;
    SS1 DYNAMIC BIND SS3;
    SS1 DYNAMIC BIND IDC1;
    SS3 DYNAMIC BIND IDC1 AND IDC2;
    SS2 DYNAMIC BIND IDC2;
  } // Manager’s attachments
  Manager1 DYNAMIC BIND IDC1 AND IDC2;

  // attachments of KDC1
  KDC1 DYNAMIC BIND IDC1;

  // attachments of IDC1 and IDC2
  IDC1 DYNAMIC BIND IDC2;
  IDC1 DYNAMIC BIND Client1 AND Client2 AND KDC1 AND HC1 AND SS1 AND SS3;
  IDC1 DYNAMIC BIND Manager1;
  IDC2 DYNAMIC BIND Client3 AND Client4 AND Reserved1 AND SS2 AND SS3;
  IDC2 DYNAMIC BIND Manager1;

  // attachments of HC1
  HC1 DYNAMIC BIND IDC1;

  // attachments of Reserved1
  Reserved1 DYNAMIC BIND IDC2;
In this specification, we have instantiated all participating components and defined connections between them. In the next section, we specify security properties of these components and demonstrate how they can be reasoned.

### 7.4 Security properties

There are several constraints expressed through the use of security properties, i.e., if software components do not satisfy system security requirements then these components are not allowed to connect to the SNS. In this section, we demonstrate how connections among system components can be established and governed using security properties introduced in details in the previous chapters. The software components utilised in this section have been presented in Chapter 3 and the previous sections of this chapter. We describe ‘getSharedKey()’ functionality which allows different components to get shared keys, and consequently communicate securely between each other. Besides, we present other functionalities including ‘storeData()’ (storing the system and users’ data) provided by SSs and ‘organiseConnection()’ (creating secure channels) supported by every component.

#### 7.4.1 Distribution of shared keys

KDCs provide ‘getSharedKey()’ functionality to allow various components, e.g., IDC1 and IDC2, to communicate securely. More specifically, this functionality is needed for distributing symmetric cryptography keys (shared keys) used to secure communications inside the system and between clients and the system. Users (clients) and system components make requests to KDCs for shared keys to be used with other clients or the system. Shared keys can be used as one time keys. Also, we suppose that all pairs of private/public cryptographic keys have been distributed securely.

For example, KDC1 (KDC1 and KDC2 both generate the same shared keys because they are synchronized) provides functionality ‘getSharedKey(Key, IDC1, IDC2)’ that is used by IDC1 and IDC2 in order to get shared keys and use them to secure communications between each other. KDC1 also provides this functionality ‘getSharedKey’ to other pairs of components including SS1 and SS2, SS1 and SS3, SS3 and SS2, IDC1 and SS3, IDC1 and SS1, IDC2 and
SS3, IDC2 and SS2, IDC1 and Client1, IDC1 and Client2, IDC2 and Client3, IDC2 and Client4, IDC2 and Client5, Manager1 and IDC1, Manager1 and SS1, Manager1 and SS3, Manager1 and SS2, Manager1 and IDC2, Manager1 and Reserved1, IDC1 and HC1, IDC2 and HC2. Security properties of these different components are similar to security properties described below. Furthermore, KDC1 supports the 3DES algorithm from 10am till 5pm every day with probability of 100%. However, from 5pm till 10am of the next day there is probability of 10% that the system and its network will be overwhelmed by the a large number of users from China and that KDC1 will not be able to provide shared keys with probability of 10% at that period of time. On the other hand, other components such as IDC1 and IDC2 know about this issue, and hence, require the provided security properties all the time but with probability of at least 90%.

Ensured and required security properties related to functionality ‘getSharedKey()’ provided by KDC1 for IDC1 and IDC2 and specified in GIZKA.ESCL are illustrated in Figure 7.5.

```java
INTERFACE KDC1 {
  VAR {
    SASO.SecurityAlgorithm.SAlgKeyExchange.IKE ALIAS IKE;
    SASO.SecurityAlgorithm.SAlgKeyExchange.DeffieHellman ALIAS DeffieHellman;
    SASO.SecurityAlgorithm.SAlgEncryption.SAlgSymmetric.3DES ALIAS 3DES;
    SASO.SecurityConcept.SConProtocol.SConSecureCommunication.SSL ALIAS SSL;
    SFO.SecurityEntity.KEY.SymmetricKey ALIAS SymK; //symmetric (shared) key
    SFO.SecurityEntity.KEY.KeySize ALIAS KeySize=168;
    SFO.SecurityEntity.KEY.PrivateKey ALIAS PriK; //asymmetric cryptography private key
    SFO.SecurityEntity.KEY.PublicKey ALIAS PubK; //asymmetric cryptography public key
  }
  FUNCTIONALITY {
    REQUIRED {...}
    PROVIDED { STRING getSharedKey(SymK,IDC1,IDC2); }
  }
  SECURITY {
    PROVIDED getSharedKey(SymK,IDC1,IDC2) {
      KeyAlgorithm(SymK[IDC1,IDC2],3DES,SSL).
      KeySize(SymK[IDC1,IDC2],KeySize).
      Signed(SymK[IDC1,IDC2],PriK[KDC1]).
      Owned(PubK[KDC1],KDC1).
      Owned(PriK[KDC1],KDC1).
      TrustedChannel(KDC1,IDC1 AND IDC2,SSL).
      KeyGenerated(SymK[IDC1,IDC2])—
        KeyAlgorithm(SymK[IDC1,IDC2],3DES,SSL), KeySize(SymK[IDC1,IDC2],KeySize).
      Shared(SymK[IDC1,IDC2],IDC1,IDC2)—KeyGenerated(SymK[IDC1,IDC2]).
      KeyDistribution(SymK[IDC1,IDC2],IKE,DeffieHellman).
      TrustedPath(KDC1,IDC1 AND IDC2)—TrustedChannel(KDC1,IDC1 AND IDC2,SSL).
      IF(Time(<10am,5pm>)) { Probability([1]) }
      IF(Time(<5pm,10am>)) { Probability([0.9,1]) }
      ELSE { Probability([1]) }
    }
  }
}
```

Figure 7.5. Security properties of KDC1 for IDC1 and IDC2
First, we declare several constants using the keyword `VAR` in order to specify security algorithms and entities from our security ontologies (the security algorithm-standard ontology (SASO) and the security function ontology (SFO)) including 3DES, SSL, IKE, Diffie-Hellman, and symmetric and asymmetric cryptography keys. Second, we express the provided functionality ‘getSharedKey’ using the keyword `FUNCTIONALITY`. Third, we use the keyword `SECURITY` to declare security properties. The security function ‘KeyAlgorithm()’ defines the shared key ‘SymK’ generated with 3DES and SSL algorithms. Besides, the functions ‘KeySize()’ and ‘KeyGenerated()’ express that this key is generated for IDC1 and IDC2 if the algorithm 3DES is applied with the key size of 168 bits. The property ‘Signed()’ ensures that the shared key is digitally signed using the KDC1’s private key. Two security properties ‘Owned()’ express the fact that asymmetric cryptography public and private keys are owned by KDC1. Another property ‘Shared()’ is ensured if the required property ‘KeyGenerated()’ is satisfied.

The next property ensures a trusted channel is established among KDC1, IDC1, and IDC2 in order to secure communications among them. The next rule states that a trusted path is established among components if a trusted channel is created among KDC1, IDC1, and IDC2 using the SSL protocol. Finally, we define that security properties are valid with probability of 100% from 10am till 5pm and valid with probability of 90% from 5pm to 10am of the next day.

Required and ensured security properties for IDC1 involved with functionality ‘getSharedKey()’ are shown in Figure 7.6.

```plaintext
INTERFACE IDC1 {
    VAR {
        SASO.SecurityAlgorithm.SAlgKeyExchange.IKE ALIAS IKE;
        SASO.SecurityAlgorithm.SAlgKeyExchange.DiffieHellman ALIAS DiffieHellman;
        SFO.SecurityEntity.KEY.SymmetricKey ALIAS SymK;
        SFO.SecurityEntity.KEY.PrivateKey ALIAS PriK;
    }
    FUNCTIONALITY {
        REQUIRED { STRING getSharedKey(SymK,IDC1,IDC2); }
        PROVIDED { … };
    }
    SECURITY {
        REQUIRED getSharedKey(SymK,IDC1,IDC2) {
            ←Signed(SymK[IDC1,IDC2],PriK[KDC1]),
            KeyExchanged(SymK[IDC1,IDC2],KDC1,IDC1)—
                KeyDistribution(SymK[IDC1,IDC2],IKE,DiffieHellman).
            BelievesShared(IDC1,IDC2,SymK[IDC1,IDC2])—
                Shared(SymK[IDC1,IDC2],IDC1,IDC2), KeyExchanged(SymK[IDC1,IDC2],KDC1,IDC1).
            ←Probability([0.9,1]).
        }
    }
}
```

Figure 7.6. Security properties of IDC1 for IDC2 and KDC1
As in the previous example, first we define some constants using the keyword \texttt{VAR}. Second, we specify required and provided functionalities. Third, we declare security algorithms and entities from our security ontologies including IKE, Diffie-Hellman, and cryptographic keys. Then, we specify required security property ‘Signed()’ that states that the shared key should be signed by the private key of \texttt{KDC1}. Then, we specify security properties where the first ensured security property of \texttt{IDC1} states that the shared key is securely exchanged between \texttt{KDC1} and \texttt{IDC1} if this key is distributed between \texttt{IDC1} and \texttt{KDC1} using the IKE and Diffie-Hellman cryptographic algorithms. The next rule specifies that \texttt{IDC1} believes this key is shared between \texttt{IDC1} and \texttt{IDC2} if ‘SymK’ is shared between \texttt{IDC1} and \texttt{IDC2} and exchanged securely between \texttt{KDC1} and \texttt{IDC1}. The last rule states that required security properties should be valid all the time with probability of 90% and more.

As illustrated in Figure 7.7, the required and ensured security properties of \texttt{IDC2} for functionality ‘getSharedKey’ are similar to the security properties of \texttt{IDC1}.

\begin{lstlisting}[language=Java]
INTERFACE IDC2 {
  VAR {
    SASO.SecurityAlgorithm.SAlgKeyExchange.IKE ALIAS IKE;
    SASO.SecurityAlgorithm.SAlgKeyExchange.DiffieHellman ALIAS DiffieHellman;
    SFO.SecurityEntity.KEY.SymmetricKey ALIAS SymK;
    SFO.SecurityEntity.KEY.PrivateKey ALIAS PriK;
  }
  FUNCTIONALITY {
    REQUIRED {
      STRING getSharedKey(SymK,IDC1,IDC2);
    ...
    }
    PROVIDED {
      ...
    }
  }
  SECURITY {
    REQUIRED getSharedKey(SymK,IDC1,IDC2) {
      ←Signed(SymK[IDC1,IDC2],PriK[KDC1]).
      KeyExchanged(SymK[IDC1,IDC2],KDC1,IDC2)←
        KeyDistribution(SymK[IDC1,IDC2],IKE,DiffieHellman).
      BelievesShared(IDC2,IDC1,SymK[IDC1,IDC2])←
        Shared(SymK[IDC1,IDC2],IDC2,IDC1), KeyExchanged(SymK[IDC1,IDC2],IDC2,IDC1).
      ←Probability([0.9,1]).
    }
    ...
  }
}

Figure 7.7. Security properties of IDC2 for IDC1 and KDC1
\end{lstlisting}

The security properties between \texttt{IDC1} and \texttt{KDC1} regarding ‘getSharedKey’ are compatible because \texttt{IDC1} require security properties to be valid with probability of at least 90% while \texttt{KDC1} guarantees probability of 100% from 10am till 5pm and probability of 90%-100% in the rest of the time. Besides, \texttt{IDC1} and \texttt{IDC2} require that shared keys should be distributed using the IKE and Diffie-Hellman algorithms while \texttt{KDC1} supports such capability. Hence, the
required and ensured security properties of the components satisfy each other and the compositional security contract between them is formed:

\[
\text{CsC(IDC1,KDC1)} \leftarrow \text{BelievesShared(IDC1,IDC2,SymK[IDC1,IDC2])}.
\]

Likewise, CsC between IDC and KDC can be formed as in the following way:

\[
\text{CsC(IDC2,KDC1)} \leftarrow \text{BelievesShared(IDC2,IDC1,SymK[C1,C2])}.
\]

The compositional security contract between IDC and IDC is:

\[
\text{CsC(IDC1,IDC2)} \leftarrow \text{CsC(IDC2,KDC1)} \land \text{CsC(IDC1,KDC1)}
\]

These contracts deliver confidentiality and integrity and should be verified at runtime:

\[
\text{Confidentiality} \leftarrow \text{CsC(IDC1,IDC2)}.
\]
\[
\text{Integrity} \leftarrow \text{CsC(IDC1,IDC2)}.
\]

As previously mentioned, the required and ensured security properties for the same functionality provided by KDCs for other components are almost identical, hence, they are not illustrated here. It is worth mentioning that other security properties for such functionalities as storing the system and users’ data, i.e., ‘storeData()’, provided by SS1, SS2, and SS3 and organising secure channels, i.e., ‘organiseConnection()’ are presented in the next sections.

### 7.4.2 Storing data

As previously mentioned, storage servers such as SS1, SS2, and SS3 deliver several functionalities including storing the system and users’ data (‘storeData()’). Such data may include users’ credentials, credit card details, the history of users’ actions such as users’ chatting history, and so on. Besides, these SSs guarantee confidentiality, integrity, and nonrepudiation of this data. The security properties of SS1 and Client1 for functionality ‘storeData()’ are presented in Figure 7.8. It is worth mentioning that we assume that all private/public and shared keys have been successfully distributed and IDC1 is used as a proxy between Client1 and SS1.

```plaintext
INTERFACE SS1 {
  VAR {
    SFO.SecurityEntity.KEY.SymmetricKey ALIAS SymK;
    SFO.SecurityEntity.KEY.PrivateKey ALIAS PriK;
  }
  FUNCTIONALITY {
    ...
  }
}
```
Chapter 7  
A social network system as a case study

Figure 7.8. Security properties of SS1 for ‘storeData()’ functionality

The security properties of SS2 and SS3 are almost identical and are not illustrated here. If other components have such requirements as confidentiality or nonrepudiation, SSs deliver them because they encrypt and sign the stored data. For example, the security properties of Client1 are illustrated in Figure 7.9. The security properties of Client1 and SS1 satisfy each other, and consequently, the compositional security contract between them is organised which delivers confidentiality, integrity, and nonrepudiation.

Figure 7.9. Security properties of Client1 for ‘storeData()’ functionality

The security properties for other clients (i.e., users) are similar.
7.4.3 Secure connections

Various components need to communicate to each other, however, it should be done in a secure way. In our SNS, IDCs provide ‘organiseConnection()’ functionality for participating components which should satisfy security properties of each other in order to communicate securely. We demonstrate security properties for the given functionality using IDC1 and IDC2 as an example. We suppose that all private, public and shared cryptographic keys have been distributed successfully among components. The security properties of IDC1 and IDC2 for functionality ‘organiseConnection()’ are illustrated below in Figures 7.10 and 7.11.

```
INTERFACE IDC1 {
    VAR |
        SFO.SecurityEntity.KEY.PrivateKey ALIAS PriK;
        SASO.SecurityConcept.SConProtocol.SConSecureCommunication.SSL ALIAS SSL;
    }
    FUNCTIONALITY {
        REQUIRED { STRING organiseConnection(PriK,IDC1,IDC2); }
        PROVIDED {...}
    }
    SECURITY {
        PROVIDED organiseConnection(PriK,IDC1,IDC2) {
            Protected(data,IDC1,IDC2).
            InternalTransfer(data,IDC1,IDC2)←Signed(data,PriK),TrustedPath(IDC1,IDC2).
            TrustedChannel(IDC1,IDC2,SSL).
        }
    }
}

Figure 7.10. Security properties of IDC1 for ‘organiseConnection()’ functionality

INTERFACE IDC2 {
    VAR |
        SFO.SecurityEntity.KEY.PrivateKey ALIAS PriK;
        SASO.SecurityConcept.SConProtocol.SConSecureCommunication.SSL ALIAS SSL;
    }
    FUNCTIONALITY {
        REQUIRED { STRING organiseConnection(PriK,IDC2,IDC1); }
        PROVIDED {...}
    }
    SECURITY {
        PROVIDED organiseConnection(PriK,IDC2,IDC1) {
            Protected(data,IDC2,IDC1).
            InternalTransfer(data,IDC2,IDC1)←Signed(data,PriK),TrustedPath(IDC2,IDC1).
            TrustedChannel(IDC2,IDC1,SSL).
        }
    }
}

Figure 7.11. Security properties of IDC2 for ‘organiseConnection()’ functionality
The security properties of IDC1 and IDC2 satisfy each other because they have almost identical security requirements and capabilities. The security properties of other components for the same functionality are similar.

7.5 Dealing with security attacks

As previously mentioned, our SNS has to deal with many security threats including security attacks such as multi-phased distributed attacks (e.g., Mitnick attacks) and different DDoS and DRDoS attacks (e.g., Smurf or Fraggle attacks). Hence in this section, we present several attack scenarios and demonstrate how the system can deal with them.

7.5.1 Dealing with Mitnick attacks

In this section, we illustrate how our SNS deals with Mitnick attacks described in Chapter 5 and 6, how these attacks are detected, analysed, correlated and anti-correlated, how the information about them is distributed among DCs, and how such security attacks are resisted and mitigated.

Overview of the Mitnick attack

As previously mentioned, the Mitnick attack is rather difficult to detect and protect systems against them. Moreover, as illustrated in Chapter 5, such attacks can be detected and mitigated if DCs operate as a coalition. We describe how our security ontologies can be utilised and demonstrate them through the use of our SNS. The Mitnick attacks can be utilised as a baseline for our SNS because if the system cannot detect these attacks then it cannot register “smarter” attacks.

This security attack exploits weakness of the TCP protocol design in making a TCP connection between two hosts (DC1 and DC2) called the three-way handshake:

4. DC1 sends a SYN packet to DC2 in order to initiate a TCP connection with DC2.
5. To establish a TCP connection with DC1, DC2 returns a SYN/ACK packet.
6. DC1 receives a SYN/ACK packet and returns an acknowledgment to DC2. This is the final stage of the handshake when a connection is established.

For example, our SNS (see Figure 7.1 and 7.2) consists of storage servers (SSs), defensive components (DCs), and clients that need to communicate securely with each other over the network. Moreover, SSs store information about clients including their credit card details and home addresses. Such information is required because the SNS provider has to be sure that its clients are real people who satisfy certain requirements (e.g., age). In our case, SSs are represented by SS1, SS2, and SS2 while clients are Client1, Client2, Client3, Client4, and Client5. Besides, there are IDC1 and IDC2 that are responsible for protecting the system. Clients connect to the system through IDCs, i.e., IDCs operate as proxies. Initially, IDC1 and IDC2 have the same configuration by default, however, they can be customised later for
supporting clients from different regions of the world. **IDC1** and **IDC2** allow clients from the U.S.A. to connect to our SNS while other DCs (not shown here) support clients from Australia. And, even if they are responsible for different geographical regions, they still need to synchronise game data for delivering availability.

An attacker (**Attacker**) wants to get credit card details of all users from the U.S.A. (**Client1**, **Client2**, **Client3**, and **Client4**). After registering as a user from the U.S.A. using stolen credentials and social engineering skills, after analysing network traffic, **Attacker** figures out that **IDC1** and **IDC2** are responsible for this region. One of the ways for **Attacker** to get credit card details is to perform the Mitnick attack against both **IDC1** and **IDC2**. It is essential to mention, that the classical Mitnick attack can be performed at the first stage of penetration. The next possible stage can be performing the XML injection attack against hosted Web services or the SQL injection attack against hosted databases (if there are any). Also, since the Mitnick attack is related to the man-in-the-middle attack that exploits weakness of the design of a TCP protocol in making a TCP connection (three-way handshake) between peers. We note that a combination of security attacks against different layers such as application (WS and P2P), network (DoS), and communication (Sniffing) layers is rather dangerous because of difficulties with detecting and resisting them.

![Figure 7.12. The Mitnick attacks against DC2](image)

**Figure 7.12. The Mitnick attacks against DC2**

The Mitnick attack consists of several steps, as demonstrated in Figure 7.12. The attacker (**A**) tries to attack **IDC2** through trusted **IDC1** using the TCP SYN attack based upon the three-way handshake during initiating a TCP connection:

1. To block communications between **IDC1** and **IDC2**, **Attacker** starts the TCP SYN attack against **IDC1**.
2. Then, **Attacker** sends multiple TCP packets to **IDC2** in order to predict a TCP sequence number generated by **IDC2**.
3. **Attacker** pretends to be **IDC1** by spoofing **IDC1**’s IP address and initiates a TCP session between **IDC1** and **IDC2** by sending a SYN packet to **DC1** (Step 1 of the three-way handshake).

4. Then, **IDC2** responds to **IDC1** with a SYN/ACK packet (Step 2 of the three-way handshake). However, **IDC1** cannot send a RST packet to terminate a connection because of because of many partially opened connections caused by the TCP SYN attack from Step 1.

5. **Attacker** cannot see a SYN/ACK packet from Step 4, however, **Attacker** can apply a TCP sequence number from Step 2 and **IDC1**’s IP address and send a SYN/ACK packet with a predicted number in response to a SYN/ACK packet sent to **IDC1** (Step 3 of the three-way handshake). Now, **IDC2** thinks that a TCP session is established with a trusted **IDC1**. **Attacker** can perform an attack against hosted Web services or databases (if there are any).

As it can be seen, there are some issues in the example described above. For example, if **IDC1** and **IDC2** do not cooperate and do not constantly exchange data related to security attacks, then at the initial stage of the attack, **IDC1** detects just a short TCP SYN attack but **IDC2** registers only attempts to predict a TCP sequence number and IP spoofing. So, we raise several questions:

a. How to detect such multi-phased distributed attacks as Mitnick attacks?

b. How different and seemed independent attacks such as a TCP SYN attack, a TCP sequence number prediction, and IP spoofing can be correlated in order to find the root triggering attack?

c. How DCs should collaborate with each other in order to anti-correlate, resist, and mitigate such attacks?

d. How countermeasures can be devised?

e. How information about attacks and defenses can be shared and distributed among DCs?

In the next section, we answer these raised questions.

**Scenario and analysis**

In this section, we explain how the Mitnick attack can be detected using methods described in the previous chapters and how the SNS can respond.

As illustrated in Figure 7.2, **Client3** links to **IDC1** (a part of **DC1**) in order to use our SNS. At the same time, **IDC1** actively communicates with **IDC2** and **Manager1** synchronizing data, looking after each other, and constantly exchanging information about security attacks and defenses. However, **Attacker** tries to perform a Mitnick attack (maybe in conjunction with WS attacks) against **IDC2**.
Supposing, \textbf{Attacker} attacks (from a host with an IP address 192.2.1.23) IDC2 (IP address is 147.202.46.40). IDC1’s IP address equals 147.202.46.43. Since we have previously introduced all steps of this attack, now we only present here how IDC1 and IDC2 deal with it.

At the first step of the Mitnick attack, \textbf{Attacker} needs to spoof IDC2’s IP address to initiate many TCP connections with IDC1. Consequently, IDC1 detects many partially opened TCP connections from IDC2’s IP address 147.202.46.40 at the period of time from 15.50.00 to 15.55.00 on 10.10.2007. Therefore, IDC1 sends a message (containing the class ‘\textit{SecurityEvent}’ from SAVO) to IDC2 to verify if IDC2 really has initiated all these connections. DC1 waits for three minutes and if DC2 does not respond, then IDC1 takes prompt actions, i.e., send an alert message to \textbf{Manager1}. The class ‘\textit{SecurityEvent}’ from SAVO and its data properties are specified in as GIZKA language follows:

```java
Class SAVO.SecurityEvent {
    Property NAME;
    Property TARGET;
    Property SOURCE;
    Property TIME;
}
```

As mentioned in Chapter 5, properties ‘\textit{NAME}’, ‘\textit{TIME}’, ‘\textit{SOURCE}’, and ‘\textit{TARGET}’ define the time when an event occurred, its duration, and an IP address of a target and a host from which it was initiated. The instance of the class ‘\textit{SecurityEvent}’ called ‘\textit{SecEv1}’ is specified and generated in the following way:

```java
SAVO.SecurityEvent SecEv1 = NEW SAVO.SecurityEvent {
    NAME = SAO.Attack.DoSAttack.DDoSAttack.TCPSYNAttack;
    TARGET = 147.202.46.43;
    SOURCE = 147.202.46.40;
    TIME = 15.50.00 10.10.2007 - 15.55.00 10.10.2007;
}  
EMIT SecEv1;
```

Then, IDC1 sends a message containing ‘\textit{SecEv1}’ in its body to IDC2 and Manager1. If IDC2 does not reply after five minutes, then IDC1 generates an alert (another instance of ‘\textit{SecurityEvent}’) and sends it to Manager1 (for reliability Manager1 also generates an alert). At the same, IDC2 registers that someone from the IP address 190.2.1.23 tries to predict a TCP sequence number. This event (also sent to and analysed by Manager1) is specified as:

```java
SAVO.SecurityEvent SecEv2 = NEW SAVO.SecurityEvent {
    NAME = SAO.Attack.TCPSequenceNumberPredictionAttack;
    TARGET = 147.202.46.40;
    SOURCE = 192.2.1.23;
    TIME = 15.50.00 10.10.2007 - 15.55.00 10.10.2007;
}
```
Then, **Manager1** compares parameters of ‘SecEv1’ and ‘SecEv2’.

```java
IF(SecEv1.NAME == SAO.Attack.DoSAttack.DDoSAttack.TCPSYNAttack AND
SecEv2.NAME == SAO.Attack.TCPSequenceNumberPredictionAttack
AND SecEv1.SOURCE == SecEv2.TARGET AND SecEv1.TIME == SecEv2.TIME) {
    SAVO.SecurityEvent SecEv3 = NEW SAVO.SecurityEvent {
        NAME = SAO.Attack.MultiPhasedDistributedAttack.MitnickAttack;
        SOURCE = 192.2.1.23;
        TARGET = 147.202.46.40;
        TIME = 15.50.00 10.10.2007 - 15.55.00 10.10.2007;
    }
    EMIT SecEv3;
}
```

Because both **IDC2** and **Manager1** get ‘SecEv1’ and send the verification request to **IDC1** but in turn **IDC2** and **Manager1** do not receive any acknowledgement from **IDC1**, then **Manager1** concludes that the Mitnick attack is occurring and broadcasts ‘SecEv3’ to other managers and DCs (not illustrated here). It is necessary to mention that different security attacks are defined by rules (similar to expert-based systems) stored by **Manager1**. In the case of the Mitnick attack, **IDC1** registers a lot of partially opened connections from **IDC2**. At the same time (the time period of five minutes), **IDC2** detects many attempts to predict a TCP sequence number. Based on this information, **Manager1** concludes that all these attacks are related to each other and performed by one attacker (**Attacker**).

These attacks should be not treated as independent attacks but as parts of more complex coordinated multi-phased distributed attacks in order to observe and control the whole root attack. Such knowledge about the occurring attack allows to develop countermeasures faster and in a more comprehensive way.

Further, after the attack has been identified a group of SNS company’s software developers and system administrators start developing and deploying countermeasures. Well known attacks can be dealt automatically using the signatures of security attacks. However, new attacks will need interference of humans and software agents with artificial intelligence capabilities (e.g., case-based systems or artificial neural networks) to analyse attacks and create defenses.

After a countermeasure is found, it is distributed among other DCs and managers that belong to the SNS. The process of development and deployment of a countermeasure is not shown here, only the process of sharing a countermeasure is demonstrated.

One of the ways, to stop this attack is to close all connections from the spoofed IP address (147.202.46.40), then increase the size of the connection queue (+50), the decrease the time-out waiting for the three-way handshake (-1 millisecond), and finally block all connections (block...
all ports) from the Attacker’s IP address (192.2.1.23). The process of attack correlation and deployment of countermeasures is illustrated as follows:

```plaintext
IF(SecEv3.NAME == SAO.Attack.MultiPhasedDistributedAttack.MitnickAttack) {
    SDO.MitnickDefence MitnickDefence_1 = NEW SDO.MitnickDefence {
        ACTIONS {
            CLOSE_CONNECTION_IP = 147.202.46.40;
            BLOCK_IP = 192.2.1.23;
            INCREASE_CONNECTION_QUEUE = +50;
            DECREASE_TIMEOUT = -1ms;
        }
        ENFORCE WSMitnickDefence_1;
    }
}
```

Security ontologies also can be a target for the attackers, hence, they should be secured. Thus, a modified version of security ontologies is securely distributed through the use of encryption algorithms such as the AES algorithm or 3DES (for delivering confidentiality). It is also signed and time-stamped for providing integrity, authentication, and non-repudiation using RSA and SHA-256. In the next section, we demonstrate how the SNS deals with flooding DoS attacks.

### 7.5.2 Dealing with flooding DoS attacks through structural reconfiguration

As mentioned in Chapter 5, various flooding DoS attacks including Smurf and Fraggle attacks cannot be stopped by traditional security mechanisms. There is no a fundamental defence against them because of their nature. These security attacks exploit normal functionalities of the Internet, i.e., if the victim blocks packets from any key Internet systems such as DNS servers then she may also excise herself from some parts of the Internet. Hence, we propose a new mechanism to withstand such attacks through dynamic structural reconfiguration and adaptation of the system. In this section, we demonstrate it and show how reconfigurations of the SNS can be defined, as illustrated in Figure 7.13. Besides in the previous sections, we have presented the style ‘PeerStyle’ and specified our SNS using GIZKA. The full description can be found in the previous sections.

```plaintext
SYSTEM SNS : PeerStyle = {
... //communications should be protected through enforcing the system layer patterns
    ENFORCE SPO.SystemLayer.SecureCommunication(AllComponents);
    ENFORCE SPO.SystemLayer.SingleAccessCheckPoint(IDC);

    //in the beginning SuperP2PPattern should be enforced
    ENFORCE SPO.SystemLayer.SuperP2PPattern.AllowedSuperP2PConnection(ClientT,IDCT);

    //if there is the Smurf attack then PureP2PPattern should be enforced
    IF(SecurityEvent1.NAME == SAO.Attack.DoSAttack.DRDoSAttack.SmurfAttack) {
        ENFORCE SPO.SystemLayer.PureP2PPattern.AllowedSameP2PConnection(ClientT);
    }
}
```
RELATIONS {
ATTACHMENTS {
    COMPONENT IDC3 : PeerStyle.IDCT = new PeerStyle.IDCT {...};
    Client3 DYNAMIC BIND IDC3;
    Client4 DYNAMIC BIND IDC3;
}
}
...

**Figure 7.13. Specification of SNS**

For example, if there is the Smurf attack then the system reconfigures its structure dynamically from the pure P2P structure to the super P2P structure, i.e., clients are allowed to connect other clients while in normal regime they are forbidden to do this. Following such strategy, it is possible to clients still use the system through other clients. In our case, IDC2 is under attack, hence IDC3 is deployed to support IDC2 and allow Client3 and Client4 to use the system. Also, security properties are checked at runtime. If there is the Mitnick attack then the system uses mechanisms previously described including blocking the attacker’s IP address, increasing the connection queue, and decreasing timeout.

As mentioned in Chapter 6, the runtime reconfiguration should not violate the programmed reconfiguration because dependencies at design time should not be violated by changes at runtime. However, it is rather difficult to manage integrity of conformance of the programmed and runtime reconfigurations since runtime reconfigurations cannot be described at the ADL level. Programmed reconfigurations are specified in GIZKA and capture changes known at design time (ADL level). They utilise ‘IF-ELSE’ conditions at the ADL level to express the system’s changes that can be foreseen. However, programmed reconfigurations are not capable to handle undefined changes in the system and the system’s environment that may happen in future. It is worth mentioning that runtime reconfigurations depend on the actual implementations. We integrate programmed and runtime reconfigurations as follows:

The programmed reconfiguration is specified in GIZKA as previously described and converted to the ADL-level script. After this, the script is utilised to manage and control the system at runtime. When changes take place at runtime, this initial script is rewritten according to new introduced requirements. Further, the modified script is applied to update the initial ADL-level script which is evaluated at runtime in order to verify any inconsistencies with the ADL-level script. Thus, runtime reconfigurations include the script modifications and the actual structural system reconfigurations.
7.6 Summary

In this chapter, we have presented the social network system (SNS) as a case study that adopts our approach. We have described the business plan for this SNS to show its feasibility and applicability to the real world. Our SNS employs the SECROBAT reference architecture in order to provide various functionalities such as chatting, message exchange, or file sharing. Because of employing SECROBAT, our SNS can withstand various computer pathogens and security attacks, especially multi-phased distributed attacks. As previously mentioned, such attacks can be identified and mitigated only through collaboration of constituent components of the SNS. To share information about these attacks and countermeasures, this SNS employs security ontologies. Moreover, various Distributed Denial of Service (DDoS) or Distributed Reflective Denial of Service (DRDoS) (e.g., Smurf and Fraggle attacks) cannot be resisted using traditional approaches such as blocking particular hosts or forbidding certain types of packets. Hence, our SNS adopts another approach that allows the system to reconfigure its architecture in order to survive during these attacks. These reconfiguration strategies and mechanisms as well as security properties of the system and its components are specified in our GIZKA language. It is necessary to mention that in this case study we have utilised only the SNS; however, various distributed systems such multi-player online gaming systems or various e-commerce systems can also utilise our approach. The main benefit of our approach for various organisations is providing the same functionalities (that they used to provide before) but with better security and at reduced costs.
Chapter 8

A prototype of a social network system

The main goal of this chapter is to demonstrate how a prototype of a social network system (SNS) can be developed using the key ideas provided in the previous chapters including the reference architecture called SECROBAT, the security ontologies and the GIZKA language. This chapter is organised in the following way. First, we provide the general overview of our prototype (based on SECROBAT) and its constituent parts (peers). Second, we present an architecture called PROTO which consists of the microkernel and modules (components or plug-ins) and used to implement different peers. Third, we demonstrate the process of development of new peers and introduce several types of them used in the SNS and their internal architecture. Fourth, we explain how security properties can be defined and reasoned. Fifth, we present how security ontologies can be specified in OWL, describe how security attacks can be identified and how the reasoning engines reason about these attacks and countermeasures to mitigate them. Sixth, we describe how the SNS is deployed and how it operates. Seventh, we discuss what kind of security tools can be applied to simulate security attacks against our prototype and how it may deal with them. Besides, we describe some limitations related to our prototype. Finally, we summarise in the last section.

8.1 General overview

In this section, we briefly describe a prototype of our social network system (SNS) (described in Chapter 7) which is implemented (in Java [Java06] together with Groovy [Groovy07]) using the architecture called PROTO. The prototype is based on our reference architecture called SECROBAT. The detailed description of the PROTO architecture is introduced in the next sections and in Appendix D.

The SNS consists of defensive peers, a manager peer, a reserved peer, storage server (SS) peers, and client peers, as illustrated in Figure 8.1. As previously mentioned, components in our approach can be treated as peers that provide various services. Every peer is actually a software application that consists of the PROTO microkernel and the PROTO modules (components) connected to this microkernel using the PROTO message exchange mechanism. First, we describe application-independent peers such as defensive peers, a reserved peer, and a manager peer. As described in Chapter 7, the defensive peers are responsible for dealing with various aspects of security including security attacks and countermeasures.
Defensive peers may include intrusion detection components (IDCs), key distribution components (KDCs) and honeynet components (HCs). More precisely, IDCs protect connections between users (client peers) and the system (SS peers and defensive peers), serve as proxies for them, detect cases of intrusion, and study log files. KDCs are responsible for distributing cryptographic keys and digital certificates used to secure different pieces of the SNS. HCs are utilised to study attacks and attract intruders by exposing well-known vulnerabilities, and consequently gather the data regarding security attacks, which is then shared with IDCs. The SNS interacts with the external world through the use of IDCs. The reserved peer is utilised if the SNS needs additional resources (e.g., during Smurf or Fraggle attacks).

The manager governs the system and its constituent components and decides when the system should change its structure and when components and peers should be connected or disconnected from the system. Moreover, the manager can be represented by a software agent or a human (e.g., a system administrator). It is worth mentioning that for simplicity we employ only one manager peer. However, it can be the main target for the attackers and the main bottleneck in the real SNS. To solve this issue, the manager peers can be organised as a supernode network placed behind defensive peers used as proxies between the manager peers and the external world.

Second, we present application-dependant peers including storage server (SS) peers which are capable to store different system data (e.g., security policies), personal users’ data, system security properties and policies, etc. For better reliability, several storage servers are employed to make the data kept on them available all the time. The data is mirrored, therefore, each storage server has a synchronised copy. Users can connect them through the use of IDCs.
Client peers are used by users to connect to the SNS and employ it, e.g., for chatting. All connections among the system users are encrypted. Clients can connect to other clients directly if the system allows it (e.g., during Smurf attacks). Also, if the SNS provider needs more resources, the system can try to distribute load dynamically and evenly. To minimize expenses, the SNS provider can utilise client peers as storage servers, however, system data stored on these clients should be encrypted.

In the next sections, we describe the PROTO architecture and the implementation architecture of the peers.

### 8.2 Peer architecture

In this section, we describe the architecture called PROTO which is used to implement a prototype of the social network system (SNS). More specifically, PROTO is utilised to develop peers which are the constituent parts of the SNS. PROTO consists of a microkernel and modules (or components) that deliver different functionalities and security features, as depicted in Figure 8.2. Besides, the PROTO architecture has several innovative features including a secure protocol, a microkernel, an internal language for developing and integrating software components (modules) and systems, a mechanism of messaging, a method for developing new software modules, and few others. Besides, there is a number of ready to use modules.

![Figure 8.2. Peer architecture](image)

More specifically, each peer developed using the PROTO architecture includes the microkernel called ‘Core’ and plug-ins (modules) which are subdivided into three layers including user interface layer (UIL), processing layer (PRL), and gateway layer (GWL) and which are connected to the ‘Core’. The difference among different peers (actual instances of PROTO) is the set of components (modules) that can be plugged in to the ‘Core’ microkernel. Moreover, the microkernel architecture (similar to MacOS X/Darwin or Hurd) allows to develop software
applications in a fast and simple way by inserting or removing modules. The separation of the modules encourages development of UIL, PRL, and GWL modules independently from each other. The detailed description of PROTO is provided in Appendix D.

8.2.1 The microkernel

The ‘Core’ microkernel consists of three components, as illustrated in Figure 8.2, including the ‘Controller’, the ‘Planner’, and the ‘Queue’. The ‘Controller’ coordinates the work of all other microkernel components performing such operations as starting and stopping the system, inserting and removing modules, showing debugging information etc. On the other hand, the ‘Planner’ supports an ordered list of modules connected to the ‘Core’, selects messages from the ‘Queue’ according to some criteria and then transfers these messages to the modules signed to work with this particular message type for further processing. The ‘Queue’ contains messages (received from users, networks, or processes) with various system flags including type, priority, bandwidth, and so on in order to easily handle various types of data. The detailed specification of the microkernel and its constituent components is presented in Appendix D.

8.2.2 PROTO modules

Modules (also called components or plug-ins) are special components that deliver various functionalities and security. New software applications with new features can be designed and developed by combining the microkernel with modules (existed or newly developed), i.e., new features can be added to such software applications by adding new modules. There are three major types of modules including:

- User interfaces modules (UIL) are employed to interact with users (through a mouse, keyboard, microphone, etc.) and interpret and visualize packets from the ‘Queue’ and then encapsulate them back.

- Processing modules (PRL) interact with external processes (e.g., system events). They receive packets from the ‘Planner’, process them and then place the processed packets into the ‘Queue’. Examples of these modules are:
  - Encryption and decryption modules which are used to secure messages;
  - Large files transfer modules which break large files into several smaller chunks;
  - Compression modules which employ compression algorithms to decrease the size of the transferred messages;
  - File sharing modules, storage and database modules etc.

- Gateway modules (GWL) listen to the network and place messages from different networks onto the ‘Queue’ as well as receive packets which have to be sent from the ‘Planner’ to the network. Each software application with such gateway can operate as a server/gateway to
other networks (e.g., distributed counting networks, file sharing networks, chats, instant messaging networks etc).

The detailed specification and the examples of the different modules are given in Appendix D.

### 8.3 Peers

If software developers adopt our approach for developing new peers, they only need to design and implement new PROTO modules (i.e., components using specified syntax) with required functionalities or utilise existing modules and plug them into the microkernel developed by us in order to create new peers (i.e., software applications) with prescribed features. If the reader is interested, the detailed description of this process can be found in Appendix D.

In the remainder of this section, we describe the actual internal architecture of peers used in our SNS.

#### 8.3.1 Defensive peers

In this section, we present the architecture of the defensive peer and tools technologies used to implement it. Besides, we describe Snort-based sensors capable to sniff and block network traffic. The conceptual topology is illustrated in Figure 8.3.

![Figure 8.3. Defensive peer](image)

The typical defensive peer consists of the core (described in the previous section) and modules (components) connected to this core. The operational process of defensive components
Chapter 8

A prototype of a social network system

including intrusion detection (IDC), honeypot (HC), and key distribution (KDC) has been already described.

Every defensive peer has the ‘Admin’ module which is used to control the peer by the manager. The ‘Storage’ module allows to connect to various databases (e.g., MySQL [MySQL07]) or other types of storages (e.g., disk files) in order to retrieve the data regarding security attacks and defenses as well as security properties. The detailed description of this module is provided in Appendix D. The main task of the ‘Cutter’ module is to split large files in smaller chunks or to combine splitted chunks to the original files. This module is used to optimise transmission of large pieces of data over the network. Another module, ‘Crypto’, is responsible for encrypting and decrypting (using different cryptographic algorithms) of data and connections inside the peer as well as outside. Besides, every defensive peer has the ‘Gateway’ module is capable to capture and parse messages and used as a proxy server between other modules and the external network.

The IDC uses Snort [Snort07] sensors that collect alert data and forward it to the MySQL database while Basic Analysis and Security Engine (BASE) [BASE07] utilises PHP [PHP07] and other tools to provide front-end for the manager (introduced further) to query the database from a Web browser. Since we deploy our sensors on Windows based systems, hence, we utilise several software packages and technologies that are specific for this particular operational system (for Linux and other *nix systems tools can vary). These packages include Apache Web Server [Apache07], the BASE web-based security console [BASE07], PHP [PHP07], MySQL [MySQL07], ADODB [ADODB07], Snort [Snort07], WinPcap [WinPcap07], SnortSam [SnortSam07] and 8Signs firewall [8Signs07].

We deploy the Snort sensors together with SnortSam and the 8Signs firewall on the Windows based systems (but Linux can be used as well). The 8Signs firewall is employed as a part of the reconfiguration process because it can block and lure attackers. More specifically, the 8Signs firewall is a stateful inspection firewall which can determine if packets can pass through the firewall based on the protocol, port, and source and destination addresses. Besides, it allows trap attackers by creating and using tarpits (a trap for malicious users). In order to leave ports scanners and attackers stuck a long period of time, tarpits accept TCP connections but never reply and ignore disconnect requests. The 8Signs firewall also is capable to ban IP addresses to prevent unwanted connection attempts, filter by MAC address, protect against TCP SYN attacks etc. The description of the software packages described above can be found in Appendix D.

8.3.2 Manager peers

The internal architecture of the manager peer is illustrated in Figure 8.4. Such type of peers is fully responsible for governing the whole SNS and its constituent components. Similar to defensive peers, it consists of the core and plugged-in modules including the ‘Storage’ module,
the ‘AdminController’ module, the ‘SecPropReasoner’ module, the ‘ProbReasoner’ module, the ‘TimeReasoner’ module, the ‘AttDefReasoner’ module, and the ‘Admin’ module. The functionalities provided by the ‘Gateway’, ‘Storage’, ‘Admin’, ‘Cutter’, and ‘Crypto’ modules have been previously introduced. However, in such a configuration the ‘Storage’ module is also utilised to connect to various databases and update date from configurations scripts specified in GIZKA which store the specific data including security properties, the rules of correlation and anti-correlation, reconfiguration strategies, and so on, and the ontology repository which contains our security ontologies. The manager uses the ‘AdminController’ module to manage other peers through the use of the ‘Admin’ modules, which are installed on all peers. This module also allows to deploy various countermeasures such as reconfiguration and adaptation strategies. Besides, the manager can order to bind or unbind the particular modules or peers, create or destroy them, block attackers using the ‘AdminController’ module and BASE, reconfigure the system using reconfiguration scripts defined in GIZKA and stored on the manager peer, allow users to communicate to each other directly or through the system, and so on.

The ‘ProbReasoner’, ‘TimeReasoner’, ‘SecPropReasoner’ and ‘AttDefReasoner’ modules are capable to reason about probability, time, and security properties and correlate and anti-correlate security attacks and defenses. The architectures of these reasoners are introduced in the following sections.

### 8.3.3 Reserved peers

The internal architecture of the reserved peer contains the core and several modules plugged-in to the core including the ‘Storage’, ‘Admin’, ‘Cutter’, ‘Crypto’, and ‘Gateway’ modules which have been already presented. If the system needs an additional peer of the certain type then the manager copies the required copy of the peer from the SS peer to the reserved peer using the ‘Storage’, ‘Admin’, ‘Cutter’, and ‘Crypto’ modules from both peers. Then, the manager executes the copied peer and leaves the existed version of the reserved peer untouched.
8.3.4 Storage server peers

Storage server (SS) peers are used to store personal users’ and system data. Several SS peers are employed for redundancy, i.e., the data is mirrored and each storage server has a synchronised copy. As previously mentioned, they are connected to each other directly. The typical SS peer consists of the core and connected to the core modules including the ‘Storage’ module, the ‘Admin’ module, the ‘Crypto’ module, the ‘Gateway’ module, and the ‘Cutter’ module. Their functionalities have been already described.

8.3.5 Client peers

Client peers are software applications utilised by users to interact with our SNS. The basic configuration includes the core and connected to it modules including the ‘Chat’, ‘Share’, and ‘Login’ modules which are application-dependant. The main purpose of the ‘Chat’ module is to allow users to chat to each other. The ‘Share’ module is used by users to share files while the ‘Login’ module provides a capability to users to log into the system using their credentials such as a pair of a login and a password. Other constituent modules including the ‘Storage’, ‘Admin’, ‘Crypto’, ‘Cutter’, ‘Gateway’ modules have been already described. It should be mentioned that there is a number of additional modules (not fully implemented and tested) that allow users to organise video and audio conferences, play online games, and collaboratively browse the Web.

8.4 Specifying security properties

In this section, we describe several reasoners used to reason about security properties including the probabilistic reasoner (‘ProbReasoner’ module), the time reasoner (‘TimeReasoner’ module), and the reasoner of security properties (‘SecPropReasoner’ module), as illustrated in Figure 8.5. The reasoning is done in a semi-automatic way, i.e., if the security properties match perfectly then the connections between components or peers are organised automatically, otherwise, the manager may decide to allow or disallow them to communicate to each other.

8.4.1 Reasoners of probability, time and security properties

As previously described, the PROTO architecture allows to build various types of software applications including component-based software systems. However, these software components (or peers) may have security properties which are time and probability dependant and which should satisfy each other. PROTO supports this feature through the use of the mechanism of security contracts utilised by software components represented in GIZKA.ESCL. These properties are stored as files associated with the particular components.

For example, when a component-requestor (REQU) wants to communicate with another component (component-responder) (RESP), a secure session should be organise. REQU first sends an initial message with its properties (probability, time, and security) to the manager and a
request message to \textit{RESP}. Then, the manager sends a request message to \textit{RESP} and waits for the response message from \textit{RESP} containing its properties. After getting this message the manager verifies using the ‘\textit{ProbReasoner}’, ‘\textit{TimeReasoner}’, and ‘\textit{SecPropReasoner}’ module if the properties of \textit{REQU} and \textit{RESP} satisfy each other. If they satisfy then a session is organised and components can communicate using messages. Besides, if security properties of one of the participating components change or network configuration changes then a new session has to be established again using the algorithm described above.

More specifically, the probabilistic reasoner is supported by the ‘\textit{ProbReasoner}’ module used for reasoning about probability properties. During this process \textit{REQU} ensures that a component responder (or several responders) (\textit{RESP}) provides a required level of probability in delivering certain services. As previously mentioned, if \textit{REQU} requires a service from \textit{RESP} with probability \([p_1,p_2]\) and \textit{RESP} delivers this service with probability \([p_3,p_4]\), then a match occurs if \(p_3 \geq p_1 \geq 0\) and \(p_4 \geq p_2 \geq 0\) where \(p_2 \geq p_1\) and \(p_4 \geq p_3\). If there is no match then negotiation is finished, otherwise this step leads to the next step on which time properties are verified by the time reasoner supported by the ‘\textit{TimeReasoner}’ module. More specifically, \textit{REQU} requires support for certain services at certain periods of time while \textit{RESP} also delivers them at certain periods of time. Several cases, as described in Chapter 6, are possible including the situations when time
periods of \textit{REQU} and \textit{RESP} concur exactly, when the certain time period of \textit{RESP} is greater than \textit{REQU}’s time period, when the time period of \textit{REQU} is greater than \textit{RESP}’s time period, and when the time periods of \textit{REQU} and \textit{RESP} partially intersect.

Reasoning about security properties is performed by the ‘SecPropReasoner’ module based on the reasoning engine and its architecture presented below. The process of reasoning regarding security properties expressed in GIZKA.ESCL can be performed through the use of the reasoning engine based on the work [Kha05]. The architecture of the reasoning engine follows the pipe-and-filter architectural style. Every connector is considered a pipe that carries a stream of characters from one filter to another. This process is illustrated in Figure 8.5.

First, the pre-processor identifies components with the common functionality on which a composition should be established. Then, security property handlers of both requestor (focal) and responder (candidate) components transfer security properties to the ‘Merger’ (the part of the ‘SecPropReasoner’ module) that merges them to one merged file. Then, the merged file goes to the reasoning engine as an input.

For example, a storage server such as \textit{SS1} introduced in the previous chapter deliver functionality ‘\textit{storeData()}’ for \textit{Client1}. The security properties for this functionality have been introduced in the previous chapters. They are stored in a script file and formatted for the reasoning engine in the following way:

\begin{verbatim}
Encrypted(data, SymK).
StorageIntegrity(data).
Signed(data, PriK).
Protected(data, SS1, Client1).
InternalTransfer(data, SS1, Client1):- Signed(data, PriK), TrustedPath(SS1, Client1).
TrustedChannel(SS1, Client1, SSL).
Confidentiality: :- Encrypted(data, SymK), TrustedChannel(SS1, Client1, SSL).
Integrity:: StorageIntegrity(data), TrustedChannel(SS1, Client1, SSL).
Nonrepudiation:: Signed(data, PriK).
\end{verbatim}

The reasoning engine uses a logic programming tool called \textit{smodels} [NS00]. \textit{Smodels} is usually used in conjunction with another tool called \textit{lparse} [Syr00] which syntactically prepare security properties for \textit{smodels} that performs actual reasoning. For example, in our case ensured and required security properties are merged to one single script file. Then, this file goes as an input to \textit{lparser} that produces a numerical representation of this merged file, i.e., transforms security properties in a way acceptable by \textit{smodels}. On the final step, \textit{smodels} takes this file as an input, reasons about security properties, and provides the processing result file as an output. Then, the processing result file is forwarded to the ‘Output validator’ that shows results or an error.

It is necessary to mention that \textit{lparse} and \textit{smodels} can be launched on Windows 2000 base system only while we use Windows XP as an implementation platform (and Linux Backtrack2). Hence, \textit{lparse} and \textit{smodels} are lunched as servers on the remote Windows 2000 machine while
the reasoning engine sends a message containing security properties and gets the results if securities properties are valid or not.

8.5 Security attacks and defenses

In the previous sections, we have presented the PROTO architecture and explained how to develop peers (software applications) for our SNS using this architecture. However, to share information regarding security attacks and defenses among various peers implemented using PROTO, we need to utilise security ontologies. More specifically, these ontologies are used to collaboratively identify security attacks, select proper countermeasures, and consequently enforce them.

Hence, in this section, we first demonstrate the process of specifying security ontologies in OWL-DL [OWL07] using the Protégé ontology editor [Protege07]. The detailed description of how to utilise Protégé as a development tool and its various plug-ins including Jambalaya or OWLViz, which help to illustrate our security ontologies, can be found in [Protege07]. We define all security ontologies described in the previous chapters including the security asset-vulnerability ontology (SAVO), the security algorithm-standard ontology (SASO), the security function ontology (SFO), the security attack ontology (SAO), and the security defence ontology (SDO). However, in this section we illustrate our example only through the use of SDO.

Then, we describe how security attacks can be specified, identified, correlated and anti-correlated using security ontologies, what countermeasures can be deployed and what kind of tools can be utilised for such purposes. Moreover, we explain how attacks can be identified, and consequently reasoned using the reasoner (represented by the ‘AttDefReasoner’ module) and the rules of correlation and anti-correlation. It should be mentioned that due to some limitations we do not use the full power of the ontological approach. More precisely, we utilise only the names of the concepts from security ontologies in the rules of correlation and anti-correlation and reconfiguration scripts specified in GIZKA. Besides, the process of reasoning and decision making is semi-automatic because of the limitations of current ontological reasoners, as explained in the remainder of this section. For example, in case of simple attacks such as Smurf attacks the system can block IP addresses of attacking hosts automatically using scripts containing anti-correlation rules, described in the previous chapters. However, in case of more sophisticated attacks the manager decides and enforces defenses manually.

8.5.1 Development tools used to specify security ontologies

We download Protégé from the website protege.stanford.edu, install and launch it. Then, we create a new project and name it ‘security_defence_ontology’. After this, we can start defining our security ontologies, SDO in particular. First, as illustrated in Figure 8.6, we specify classes ‘WSDefence’, ‘P2PDefence’, ‘MultiPhasedDistributedDefence’, ‘DoSDefence’, and
‘SniffingDefence’. Then, we continue defining SDO and the final result is illustrated in Figure 8.7. We do not demonstrate the whole development process here.

![Figure 8.6. Specifying the first five classes of SDO](image1)

![Figure 8.7. The final result](image2)

We also demonstrate a screenshot of SDO depicted graphically through the use of the OWLViz and Jambalaya plug-ins [Protege07], as illustrated in Figure 8.8. Finally, we define other security ontologies and the result is illustrated in Figures 8.9 where the Jambalaya plug-in is used to depict SAVO in the nesting view.
After finishing specifications of our security ontologies, Protégé produces five `.owl` files that have a structure similar to `.xml` files. As an example, we demonstrate below the source code of the class `BandwidthDepletionAttack` (the subclass of the class `DDoSAttack` `uses` the classes `FraggleAttack` and `SmurfAttack`) from `sao.owl`, as illustrated in Figure 8.10.
Due to limited space, other ‘.owl’ files containing the security ontologies are not presented here.

8.5.2 Reasoning about attacks and countermeasures

In this section, we introduce the process of reasoning about security attacks and defenses.

To detect security attacks and then correlate them, we use Snort [Snort07] and BASE [BASE07] engines that are connected to the system through the use of the ‘AttDefReasoner’ module which is the compulsory part of the manager peer. The Snort pattern-matching engine evaluates network packets delivered by Snort sensors against the attack rules stored in snort rule files and groups them together. The latest version of the pattern-matching engine is optimised to work in high-speed networks. This pattern-matching process reduces the number of rules that Snort has to process, and consequently increases the amount of traffic that Snort can analyse in real-time. It checks whether the parameters of the attack such as the source and target IP addresses, the source and target ports, the periods of time concur in order to conclude if the attack is happening or not. If the attack is detected, then Snort emits an alert which is put to the MySQL database. Then, we can get and show this data using BASE. BASE is capable to display different alerts and attacks (depicted in red in Figure 8.11) and help to reason about their nature. As previously mentioned, in case of simple attacks the system can block certain IP addresses and disconnect malicious users and peers automatically using a special type of system messages in conjunction with SnortSam [SnortSam07] and 8Signs firewall [8Signs07]. In case of more sophisticated attacks, the manager can determine using the BASE panel and its engine that the complex attack is occurring and respond by sending commands to the system using the ‘AdminController’.
module to block IP address of attackers, disconnect malicious users and compromised peers, and so on.

Figure 8.11. BASE

The rule that defines a simple attack such as a DDOS synflood attack can be expressed in the following way using Snort:

```snort
```

If an alert emitted upon detecting such an attack, then to avoid false positives, the system sends automatically the ping requests (“ping –a”) to the attacking host in order to verify if the host is malicious. If the host does not answer back with its name then the manager blocks this particular host and enters its IP address to the black list to prevent possible attacks in future. This fact can be expressed in reconfiguration scripts as follows:

```python
if(alert == synflood) { BLOCK_IP = Attacker’s IP; }
```

Another simple attack such as a Smurf attack is specified in Snort as follows:

```snort
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Broadscan Smurf Scanner"; dsize:4; icmp_id:0; icmp_seq:0; itype:8; classtype:attempted-recon; sid:478; rev:4;)
```

As a countermeasure, the system can also block the IP addresses of the malicious hosts. In case of more complicated attacks such as WS Mitnick attacks or other multi-phased distributed attacks, the manager decides what is happening, i.e., what type of a security attack is occurring.
what should be done, and what defenses should be enforced. In our case, the manager can
disconnect the compromised peer using the ‘UNBIND’ command from GIZKA.ADL.
In the next section, we explain how the SNS can be deployed.

8.6 Deployment

In this section, we describe how to deploy and manage the SNS described above. This process
consists of a number of steps.

First, we select the particular hardware and software platform which will be used for installing
and running the SNS. Because of several issues that are out of the scope of this thesis, we have
selected Windows operational system (OS) together with VMware [VMware07] run on an Intel-
based computer. The VMware software tools allow to run virtual machines and simulate the
complex networks with many virtual hosts. More specifically, these tools allow running
multiple operating systems simultaneously on a single host computer and sharing data among
this host computer and virtual machines. Due to the lack of resources we utilise VMware to
create virtual hosts and deploy our SNS on them. In such a configuration Windows is used as a
host system for virtual machines and as the main OS for virtual hosts. It is worth mentioning
that each virtual host runs only one peer.

Second, we need to install a particular peer on a certain virtual host. We install the additional
software packages such as the Java virtual machine (JVM) [Java06] on every virtual host,
however, only instances of the defensive peer and the manager peer require other software
packages including the Apache Web Server [Apache07], PHP [PHP07], MySQL [MySQL07],
BASE [BASE07], Snort [Snort07], ADODB [ADODB07], WinPcap [WinPcap07], SnortSam
[SnortSam07] and 8Signs firewall [8Signs07]. The description of these software packages is
given in Appendix D.

Third, we modify configuration files of the Apache server, Snort, MySQL, and SnortSam. Then,
we rewrite source code of BASE and Snort for our needs and create the MySQL database (DB)
called ‘snort’ used for storing Snort alerts. We create a user ‘snort’ with a defined password and
give him/her administrative rights. We also register the Apache Web server, the MySQL
database server, and Snort as Windows services. Then, we combine and launch the 8Signs
firewall and SnortSam in order to allow them to cooperate and identify and block attacks. For
example, when a DDoS attack such as Smurf is detected, and consequently, as a part of the
reconfiguration process, the manager sends commands to the 8Signs firewall and SnortSam to
automatically block attackers. Besides, in case of complex attacks the manager can select a
specific defensive strategy, e.g., the attacked peer can be manually removed from the system till
the end of the attack and the delivery of its functionalities can be delegated to other peers of the
same type. The detailed instruction and script files are not presented here due to complexity and
dependability on a selected operational system, a firewall, an IDS, and so on.
Fourth, we install the instances of the particular system peers (i.e., defensive peers, a reserved peer, a manager peer, and SS peers) and execute them. We distribute the copies of client peers among users in order to allow them to use our SNS. It is necessary to mention that every system and client peer has a configuration file which contains various data related to the particular behaviour of each peer such as security properties defined in GIZKA.ESCL. Besides, defensive peers and the manager peer have access to the owl files (the ontology repository) containing security ontologies and configuration files containing the information about attack and defence rules (also can be stored in the DB) represented in GIZKA.ADL and GIZKA.ATTACK accordingly. These files are synchronised and stored on defensive peers and the manager peer.

As previously mentioned, we use only the names of the concepts from the security ontologies in defining the security properties, the rules of correlation and anti-correlation, and reconfiguration scripts using GIZKA.

Fifth, we execute all required peers and allow them to log into the system using their credentials such as a pair of login/password. A combination of login/IP address is used to uniquely identify a particular peer during its connection session. Then, the peers send the messages with their security properties to the manager peer which checks these properties using the reasoning engine, and consequently, decides who is allowed to be the part of the system. Besides, the manager (manager peer) represented by a human determines using scripts specified in GIZKA how the peers are composed and gives them permissions for certain actions.

Finally after all preparations, the manager can see Snort alerts and manage Snort using BASE and PROTO and decide if an attack is happening or not. The manager peer controls the system and its constituent components using the ‘AdminController’ module. The manager can order to block certain connections or peers, disconnect or restart peers, or remove a particular peer from one host and execute another peer on the same host, or allow client peers to connect to the system and each other through JXTA [JXTA04] which can be considered a pure P2P system, and so on. Snort sensors and analysers are utilised to identify and correlate attacks. We use BASE and its engine for such purposes, however, the freely available security information management (SIM) tools including OpenSIMs [OSIMs08] and Open Source Security Information Management (OSSIM) [OSSIM08] can be utilised as well to correlate the data produced by different sources. Besides, the manager peer determines what should be done in case of a particular attack and what countermeasures should be deployed.

8.7 Simulation

In this section, we discuss what tools can be possibly used to collect the data related to the network and system and simulate attacks against the prototype and how it can deal with them. To scan the network and simulate various attacks such as probing (e.g., TCP SYN scan), the Nmap scanner [Nmap07] or CORE IMPACT [CORE08] can be used. To perform distributed
denial of service (DDoS) attacks, we can utilise some tools such as Trinoo (or Trin00), Stacheldraht, Shaft, TFN (Tribe Flood Network), TFN2K, and few others [Wiki07]. It worth mentioning that there are numerous modifications of these tools because current intrusion detection systems are able to monitor traffic for DDoS tools signatures such as known character strings, known passwords, or default ports. Other security attacks such as probing and sniffing can be performed using Metasploit [MTSPLT07] or tools from a live CD Linux distribution Backtrack 2 [Backtrack2], as illustrated in Figures 8.12. This distribution focuses mainly on penetration testing and supports more than 300 security tools arranged in 12 categories such as vulnerability identification, penetration, privilege escalation, cover tracks, radio network analysis, digital forensics, and reverse engineering.

Figure 8.12. Backtrack 2

To avoid legal and ethical issues in our work, we test our SNS on virtual hosts and networks using the VMware software tools [VMware07]. Finally, to manage the Snort IDS, we employ the Basic Analysis and Security Engine (BASE) [BASE07], as illustrated in Figure 8.11. The detailed description about various software tools introduced in this section can be found in the relevant references.

8.8 Limitations

In the previous sections, we have introduced our prototype. Due to the lack of the various resources it has some limitations.

First, we do not perform quantitative and qualitative risk analysis risk analysis. Hence, we do not collect data about the network structure, installed software applications and platforms,
launched servers and their versions, hardware, active services and open ports. Moreover, we do not rank vulnerabilities and therefore do not hire experts who can make decisions regarding a trade-off between gathered security-related information and the business requirements (i.e., business requirements of the SNS provider).

Second, we do not use the full power of an ontological approach because of the problems with current reasoners which do not fully support SWRL [SWRL06], SWSL [SWSL07], or other Semantic Web rule languages. Hence, we specify the security properties using the approach proposed in [Kha05] and simply utilise the names of the concepts from the security ontologies in the security scripts. The intellectual reasoning and negotiation can be performed by a human (e.g., a manager).

Third, detection of security attacks is performed using the modified versions of Snort and BASE which were not fully integrated with the SNS. The decisions regarding the particular attack and possible countermeasures are made manually by a manager of the SNS. Only blocking of the users and attackers is performed automatically in case of simple attacks.

Fourth, our prototype has limitations related to false positives and negatives. False positives occur when an IDS registers an attack which did not happen. Sometimes, attackers try to create massive numbers of false positives to divert the attention of an intrusion system manager away from a real attack. Hence, tuning an IDS is one of the solutions to minimize false positives (acceptable rate is 60-90%) and not miss real positives. Another term “false negative” describes an IDS's inability to identify real security events, i.e., malicious activity is not detected. In an ideal case, the likelihood of having false negatives should be zero. To reduce false negatives, it is essential to understand the system's weaknesses and implementation issues. In our case, false negatives may occur due to a few factors: a) the prototype uses Snort sensors which may not register all security threats; b) peers responsible for storing security alerts may not have enough disk space to keep these alerts; c) a manager may also miss more complex attacks because of the lack of security related data or his/her level of experience.

Fifth, the failure of a component (or peer), in an ideal case, should not influence security of other components and the whole system, which is not really possible in the “Real World”. In our case, if a client component (or peer) fails then it does not disturb others. However, if a defensive component fails (e.g., IDC) during an attack and if there are no other defensive components of the same type which can perform the same tasks as the failed component then the system’s security may suffer. For example, as previously mentioned, an attacker may perform a flooding DoS against a particular IDC placed on the border between the system and the outer world. Such a case may not allow client components (peers) to continue using the system temporarily since they will need to connect to another IDC but they still will be able to use the system with some delay.
Sixth, there some issues related to the efficiency and reliability. As previously mentioned, our approach has a few limitations due to the way we have implemented the prototype. We have used P2P technologies and networks known for their excessive usage of the network traffic. Currently, peers broadcast messages to all over the P2P network. Hence, if there are too many peers (e.g., thousands) their requests and responses may overburden the network, which consequently, may become inefficient or unreliable in the worst case. However, if we utilise DHTs (distributed hash tables), previously described, then the amount of the traffic will decrease dramatically since peers will know only about a small number of their neighbours, and consequently, the system will become more structured, efficient and reliable as the number of peers starts to grow. To increase security, we suggest to use one DHT for the system peers (not shared among client peers) and one or more DHTs for clients (known to defensive peers as well).

Seventh, the reconfiguration scripts are hard-coded, i.e., if any changes are needed the manager has to modify the scripts manually.

Finally, we have not tested the SNS thoroughly. Also, we need to perform an extensive security testing (fuzzing, penetration and vulnerability testing) of the prototype.

8.9 Summary
As previously mentioned, the main goal of this chapter has been to demonstrate how the prototype of our social network system can be implemented in order to support key ideas described in the previous chapters. Hence, we have presented the prototype of our social network system built on top of the proposed reference architecture SECROBAT using the microkernel architecture called PROTO, the security ontologies, and the GIZKA language. Further, we have introduced the PROTO architecture, explained the internal architecture of peers, and demonstrated how peers used in the prototype can be implemented using Java and Groovy. Besides, we have shown the process of specifying security ontologies using OWL and Protégé. We have presented the reasoning engines used to reason about the security properties and correlate and anti-correlate security attacks and defenses. We have also discussed how security attacks can be performed against our prototype using various software tools and how it can deal with them. Finally, we have also described the limitations of our prototype.

To conclude, this prototype demonstrates that our approach presented in this thesis is applicable to the real world and valid. As previously mentioned, the architecture of this prototype and its constituent components is designed and implemented in a way that it could be further extended with additional functionalities and security features.
Chapter 9

Conclusions and further research

9.1 Summary

The key objective of this thesis was to investigate how to achieve higher-level security for component or service based software systems (CBSSs/SBSSs) through the use of an architectural approach. This investigation has illustrated that traditional information security approaches were not able to solve many types of security problems in CBSSs including identification and mitigation of security attacks such as multi-phased distributed attacks and flooding denial of service (DoS) attacks. To resolve this issue, we have proposed a new innovative solution which utilises traditional security mechanisms together with distributed detection and defenses based on dynamic structural reconfigurations and adaptations of CBSSs and their constituent components.

Software systems become highly complex and distributed because users require a lot of functionalities. This trend is driven by new technologies which use the increasing power and failing cost of network resources and new hardware. Also, software systems, and CBSSs/SBSSs in particular, have many vulnerabilities and design flaws because of the fact that system’s security is usually implemented after its functional requirements have been addressed, and consequently companies spend much time and money to fix these vulnerabilities. At the same time, a new generation of security attacks appear such as various multi-phased distributed attacks that are difficult to identify, analyse, correlate and anti-correlate. Hence, there is a strong need for a systematic software engineering approach that we call software security engineering (SSE) for designing and implementing secure and robust CBSSs/SBSSs by considering security and functional requirements at the same time. Specifically in our case, these systems are highly adaptive and reconfigurable. They consist of different constituent components such as defensive components (DCs) that collaboratively identify security attacks and deploy countermeasures. Our approach follows analogies from the human society and biological systems in which the “strong” (e.g., a defensive component) can protect the “weak” (e.g., a functional component) where the resulting relationship and the whole system become even stronger than the strongest link or the strongest individual.

In this thesis, we have first reviewed (in Chapter 2) the current issues related to software security engineering including information security, the Common Criteria, distributed
technologies such as Common Object Request Broker Architecture (CORBA), Microsoft’s .NET, Java 2 Enterprise Edition (J2EE), Peer-to-Peer (P2P), Web services (WSs), and different dynamic software architectures and approaches. In the consequent chapters (Chapter 3-6), we have reviewed autonomic computing and biological systems (served as an inspiration to our approach) and other works related to our approach such as security ontologies, security in different distributed systems, security attacks and defenses, various security specifications, architecture description languages (ADLs), security patterns, security properties, and so on. In Chapter 3, we have presented our research approach for achieving higher-level security for CBSSs/SBSSs in details. We have described our approach for managing security which includes the reference architecture called SECROBAT with defensive components and the hybrid P2P structure. Then in Chapter 4-5, we have introduced security ontologies utilised as a universal vocabulary that allows different components in SECROBAT-based systems to communicate to each other and share information about security attacks and defences. It allows them collaboratively detect attacks and deploy countermeasures. In Chapter 6, we have incorporated these security ontologies into our language for managing security called GIZKA. This language “glues” security ontologies and SECROBAT in order to specify our approach in a formal way and manage it at design time as well as at runtime. GIZKA consists of three subsets including GIZKA.ESCL, GIZKA.ATTACK and GIZKA.ADL. Briefly, GIZKA.ESCL lets define security properties (can be time and probability dependant) of software components, i.e., certain security requirements and capabilities of components should satisfy if components want to communicate, otherwise such interactions are forbidden. GIZKA.ATTACK allows to define security attacks while GIZKA.ADL specifies defenses in a formal way, hence further these attack signatures can be used to identify these attacks, correlate and anti-correlate (i.e., to select a proper countermeasure). In Chapter 7, we have demonstrated our approach through the use of the case study (a social network system). In Chapter 8, we have demonstrated how to implement the prototype and presented its main features. The prototype has not been fully implemented due to the limitation of research resources and still needs additional features, however, it has illustrated the key features of our approach.

9.2 Key contributions

In this thesis, we have presented a new architectural approach to achieving higher-level security of component (service) based software systems. This thesis has contributed to the research area of software security engineering (SSE) with the following specific key deliverables:

1. The reference architecture for managing security called SECROBAT;
2. Security ontologies as a vocabulary for sharing information among components about security attacks and defences;
3. The language called GIZKA for defining our approach in a formal way;
4. The case study with the example of the social network system and the prototype.
The significance of each key contribution is outlined below.

9.2.1 SECROBAT: a reference architecture

Our reference architecture for managing security, specifically security attacks and defenses, called SECROBAT is employed for creating secure and robust CBSSs/SBSSs that operate as a coalition of components that are adaptive and reconfigurable in order to resist security attacks. SECROBAT has the hybrid P2P (pure P2P / super-peer) structure with defensive components (DCs) which are integral parts of such coalitions and can be adapted, reconfigured, added or deleted dynamically for protecting the system and its connections from security attacks. DCs such as intrusion detection components, honeypot components, and key distribution components can be employed for resisting such security attacks as sniffing or Mitnick attacks. Besides, DCs provide traditional information security services including preventing, registering, mitigating, resisting security attacks, also tracing-back and confusing attackers, protecting components, sharing, validating, and updating cryptographic keys and security certificates. Furthermore, such coalitions evolve over the time and adapt their hybrid P2P structure dynamically whether there is an attack or a change of the environment. For example, as previously mentioned, distributed denial of service (DDoS) flooding attacks or distributed reflective denial of service (DRDoS) attacks are almost impossible to resist using traditional approaches such as to block malicious hosts or networks. Hence, we propose a solution when a coalition of components instead of only blocking sources of attacks also changes its structure from super-peer (normal regime) to pure P2P (regime of attacks) dynamically in order to mitigate security attacks and survive.

The development of our model has been inspired by the human society and biological systems including bacteria and their biofilms. Moreover, our reference architecture can be used as a reference guide for the research of the area of SSE.

9.2.2 Security ontologies

Currently, many users of software systems, specifically component-based systems (CBSSs), require more functionality, more flexibility, and better protection. Therefore, CBSSs become highly complex and increasingly distributed. At the same time, many new security attacks, especially distributed multi-phased attacks (e.g., Mitnick attacks) appear, which are quite difficult to identify and mitigate and which can be used as a baseline for measuring information security.
In this thesis, we have demonstrated that through the collaboration of system’s constituent components it is possible to detect and mitigate many security attacks. However, components should have a common vocabulary to allow them to share information with each other regarding security attacks and defenses. Hence, we have employed an ontological approach to create security ontologies that specify information security issues in a way comprehensible to both humans and software agents. We have described our security ontologies including:

- The security asset-vulnerability ontology (SAVO) – illustrates how vulnerabilities can be exploited by intruders in order to perform various security attacks against components or systems that might affect their assets evaluated by using the quantitative and qualitative analysis and protected by defensive components;
- The security algorithm-standard ontology (SASO) – specifies various cryptographic algorithms and security approaches;
- The security function ontology (SFO) – defines information security issues and is used to assist developers to create better and more efficient protection against security attacks;
- The security attack ontology (SAO) and the security defence ontology (SDO) – both express relationships among security attacks and defenses against them and closely correlate with each other. Both SAO and SDO have been utilised with other security ontologies with the purpose to provide a common vocabulary for various software components in our system in order to identify and mitigate attacks. For example, to detect any of the multi-phased distributed attacks such as Mitnick attacks, components should operate as a coalition. As long as the attack is evolving, SAO is also evolving and shared among other members of the coalition. When a new countermeasure is created by any member of the coalition, it is added directly as a new class to SDO which is then securely distributed among other members of the coalition.

9.2.3 GIZKA: a language for managing security

To formally define our approach and agglutinate its different parts, we have developed the language called GIZKA which consists of several main pieces including GIZKA.ESCL, GIZKA.ATTACK and GIZKA.ADL.

To develop secure CBSSs from software components which are usually implemented by third-parties, software integrators have to know components’ security properties. In particular, only through the knowledge of the specific security properties of the individual component, other components can we be sure that the overall system satisfies their security requirements. Moreover, after the system’s and its constituent components’ security requirements and capabilities satisfy each other and the CBSS is constructed, it should evolve and adapt consequently in order to survive during security attacks or environmental changes.
To address these issues, we have created:

- **GIZKA.ESCL** for defining the dynamic security properties of software components that are time and probability dependant;
- **GIZKA.ATTACK** for describing and identifying security attacks;
- **GIZKA.ADL** for describing countermeasures, specifying dynamic changes of the whole system and its constituent components, and achieving higher-level security.

GIZKA.ESCL incorporates such features as time, time intervals, and time sequence to state the period of time when specific security properties should be valid. It also uses probability to describe in a flexible way the security characteristics of components, where it cannot be predicted exactly which security set of security properties would be valid in future. Alternative security properties let specify different sets of security properties which can be utilised at different situations. Runtime conditions allow to express security properties in a dynamic way through the use of ‘IF-ELSE’ conditions.

GIZKA.ATTACK specifies security attacks in a formal way in order to identify, reason, and analyse them. Then, these specifications (attack signatures) are used to detect independent security attacks that are probably related to each other and analyse the root attack that may trigger these independent security attacks. Besides, GIZKA.ATTACK together with GIZKA.ADL are used for selecting and deploying (i.e., anti-correlation) the required countermeasures such as system’s reconfigurations.

GIZKA.ADL defines dynamic reconfigurations and adaptations used as a countermeasure against distributed security attacks. It also specifies connections between components and describes the process of creation and destruction of components and dependencies among them and when and how reconfigurations should take place together with a specification of what should be changed.

### 9.2.4 Case study and prototype

The main goal of developing the case study and implementing the prototype has been to support the key objectives provided in the previous chapters and show the applicability of our approach to the real world. Therefore, on the basis of our approach, we have first implemented the social network system that allows users to chat, share files, play games together, etc. An example with the social network system has been chosen to demonstrate functionality of our approach and what is more, such networks embrace very high security and trust requirements among their members.

The prototype of our social network system is built on top of the proposed reference architecture SECROBAT using the microkernel architecture called PROTO, the security ontologies, and the GIZKA language. We have presented the PROTO architecture and illustrated how the prototype
can be implemented, how security attacks can be performed and detected using different software tools, and how countermeasures can be deployed.

The prototype is implemented using Java [Java06] together with Groovy [Groovy07] utilised as a script language. Java and Groovy use the Java virtual machine (JVM), thus, our prototype is cross-platform. To define ontologies, we have utilised OWL [OWL07] while for reasoning about security properties we have used a tool called smodels [NS00] together with another tool called lparse [Syr00] that syntactically prepare data for smodels. For identifying and reasoning regarding security attacks and defences we have used Snort together with other additional tools. Such development platforms as Protégé and Eclipse have been utilised as implementation tools.

To conclude, these case study and prototype illustrate that our approach described in this thesis is applicable to the real world and valid. As previously mentioned, the architecture of this prototype (i.e., PROTO) and its constituent components is developed in a way that it could be further extended with additional functionalities and security features.

9.2.5 Conclusion

Our approach should be employed for creating and implementing secure and trusted platforms where the environment changes or the user changes the security requirements, e.g., the social network systems based on P2P technology. Our approach is able to improve their security and protect them through the use of various defensive mechanisms and dynamic reconfigurations and adaptations.

9.3 Further research

One of the directions for future research is to create more comprehensive security ontologies which will include not only concepts from software security engineering but from social engineering as well. Possibly, these ontologies will be used as a vocabulary for specifying security policies of trusted platforms (similar to IBM Trusted Virtual Domains (TVDs) [TVD07]).

Currently in our approach, dynamic reconfigurations can be governed by ‘Manager’ who can be a human or a pre-programmed software agent. In future work, artificial intelligence techniques can be investigated in order to manage complex dynamic reconfigurations at run-time that can be used to deal with security attacks with greater automation.

There is still an open challenge regarding our prototype such as performance. Hence, we plan to evaluate the performance impact of the SECROBAT architecture on the software applications.
References


References


References


References


References


References


References


238
Appendix A

Introduction to security concepts, algorithms and functions

In this appendix, we describe several security notions which have been briefly presented in Chapter 2 and 4.

A.1 SecurityAlgorithm: symmetric cryptography algorithms

In this section, we present different symmetric cryptography algorithms.

Data Encryption Standard (DES) is defined by the class ‘DES’. It is a symmetric cryptography algorithm which provides private keys with the size of 56 bits and the block size of 64 bits and which supports confidentiality. DES is used in many systems such as PGP (Pretty Good Privacy). However, it was proven that DES can be easily broken because of short keys. The class ‘DES’ is employed as a parameter for SFO security functions. For example, the functions, which state that PGP and the DES algorithm are utilised and the length of a symmetric key equals 56 bits, can be specified as follows:

```
KeyAlgorithm(SymmetricKey,DES,PGP).
KeySize(SymmetricKey,56).
```

3DES or Triple-DES (the class ‘3DES’) is a temporal solution for replacing DES with AES. 3DES simply applies DES three times. It uses private keys with the key size of 128 or 168 bits and the block size of 64 bits. As DES, 3DES supports confidentiality only. Advanced Encryption Standard (AES) was developed as a replacement of DES. It uses Rijndael block cipher and supports any block length and private keys with the key size of 128, 192 and 256 bits. Similar to most of symmetric cryptography algorithms, AES supports only confidentiality. For example, the fact that IPSec and AES are utilised together to encrypt/decrypt the data and AES provides a symmetric key with the key size of 256 bits can be illustrated in the following way:

```
KeyAlgorithm(SymmetricKey,AES,IPSec).
KeySize(SymmetricKey,256).
```
Another symmetric cryptography algorithm called Blowfish (the class ‘Blowfish’) (designed as a replacement of DES or IDEA) supports any block length and the size of shared keys up to 448 bits. Twofish (the class ‘Twofish’) has 128, 192 and 256 bits keys and the block size of 128 bits. Both algorithms support only one goal, i.e., confidentiality.

The CAST algorithm (the class ‘CAST’) has two implementations and is used by the PGP (Pretty Good Privacy) (the class ‘SecurityConcept.SConProtocol.SConEmail.PGP’) email encryption system. It has two combinations of key and block sizes: 128 bits and 64 bits or 256 and 128 bits accordingly. The class ‘CAST’, as other SASO classes, is utilised as a parameter for SFO security functions as follows:

\[
\text{KeyAlgorithm(SymmetricKey,CAST,PGP).}
\]
\[
\text{KeySize(SymmetricKey,128).}
\]

The functions above show that PGP and the CAST algorithm are employed and the length of the symmetric key equals 128 bits. Examples of the use for other symmetric algorithms are very similar.

The IDEA (International Data Encryption Algorithm) algorithm is utilised by the PGP email system. The key size and the block size are 128 and 64 bits accordingly.

Finally, there are several implementations of the Rivest algorithm: RC2, RC4, RC5, and RC6. RC2, RC5 and RC6 are examples of a block cipher while RC4 is a stream cipher. RC2 supports keys with any sizes and the block size of 64 bits. RC4 uses keys of 40 or 128 bits long. RC5 applies keys of any size and blocks of 64 bits long. RC6 is the implementation which improves RC5.

After describing symmetric cryptography algorithms in this section, in the next section we introduce security concepts related to secure communications.

A.2 SecurityConcept: secure communications

In this section, we describe secure communications, security protocols in particular, which were briefly presented in Chapter 2 and 4.

The SSL (Secure Sockets Layer) protocol is defined by the class ‘SSL’. It operates on the transport and session layers of the OSI model (layer 4 and 5) and encrypts the entire session using the TCP port 443, certificates for authentication, and the six steps handshake. The key size can be 40 bits (restricted implementations) or 128 bits (a full version). However, SSL is vulnerable to small key sizes, key compromises, and expired digital certificates. Currently, there are several versions of SSL including V1, V2, and V3. Many application layer protocols including HTTP, FTP, SMTP, IMAP, etc. can be secured by SSL. SSL V3, which is the latest version of SSL, utilises SHA-1 as a hash function, supports certificates for authentication, and
increases resistance to MITM (man-in-the-middle) attacks. Two main security objectives delivered by SSL are integrity and authentication. For example, the fact that the trusted channel between two parties Alice and Bob is organised using SSL can be express as follows:

\[
\text{TrustedChannel(Alice, Bob, SSL).}
\]

SSH (Secure Shell) (the class ‘SSH’) is used for securing remote access and remote terminal communications. It utilises the IDEA algorithm by default and the RSA algorithm for secure connection and authentication. SSH protects against sniffing, spoofing, and MITM attacks and delivers integrity and authentication.

The TLS (Transport Layer Security) protocol (the class ‘TLS’) is based upon SSL V3 but only has backward compatibility. It provides endpoint authentication and communications privacy over the Internet through the use of ports 80 or 443 and supports RSA, Diffie-Hellman, DSA (public-key cryptography), RC2, RC4, IDEA, DES, Triple-DES (3DES), AES (symmetric ciphers), and MD2, MD4, MD5 or SHA (one-way hash functions). While providing security for traffic over the transport layer, TLS does not deliver security for Web traffic at this layer. However, it prevents eavesdropping, tampering, and message forgery attacks. Two main delivered security objectives are authentication and integrity.

The HTTP/S (Hypertext Transport Protocol Secure) protocol (the class ‘HTTPS’) is used to secure HTTP traffic using SSL through the port 443.

The S-HTTP (Secure Hypertext Transfer Protocol) protocol (the class ‘SHTTP’) is utilised to secure HTTP transactions and protect individual documents and not sessions as HTTP/S. This is the main difference between HTTP/S and S-HTTP.

SET (Secure Electronic Transaction) (the class ‘SET’) is a protocol for securing credit card transactions over insecure networks. Actually, it is a set of security protocols and based upon X.509 certificates and public key cryptography to allow parties to identify themselves to each other and exchange information securely.

IPSec (IP security) (the class ‘IPSec’) is a set of protocols for securing IP communications by establishing cryptographic keys, authenticating, and encrypting IP packets. Since IPSec combines various protocols, some of them are less popular then others. For example, a combination of SSL-VPN (Virtual Private Network) is less popular because of interoperability problems. IPSec is used with IP only, however, it can be used with various VPN technologies such as L2TP (Layer 2 Tunnelling Protocol). Moreover, IPSec supports encryption and authentication mechanisms, requires certificates and pre-shared keys, and operates on the network layer (layer 3 of the OSI model). The Authentication Header (AH) protocol and the Encapsulating Security Payload (ESP) for data encryption protocol are constituent IPSec
protocols that deliver authentication and integrity. IPSec supports a transport mode for end-to-end encryption and a tunnel mode for link-to-link communications.

For example, the trusted channel between Alice and Bob can be organised using IPSec. This fact can be stated in the following way:

\[ \text{TrustedChannel}(\text{Alice}, \text{Bob}, \text{IPSec}). \]

The PPTP (Point-to-Point Tunnelling Protocol) protocol (the class ‘PPTP’) is based on the PPP (Point-to-Point Protocol) protocol and uses CHAP for authentication. PPTP supports the IP protocol only and does not encrypt data. It is considered a worse choice than L2TP.

The L2TP (Layer 2 Tunnelling Protocol) protocol (the class ‘L2TP’) supports the IP protocol as well as many others. It utilises IPSec for encryption and operates on the Data link layer (layer 2 of the OSI model).

Finally, the L2F (Layer 2 Forwarding) protocol (the class ‘L2F’) provides mutual authentication and no encryption of data. It has been combined with PPTP to create L2TP. As L2TP, L2F operates on the Data link layer (layer 2 OSI model). It delivers only authentication.

In the next section, we specify security functions in GIZKA.

A.3 Security functions

In this section, we define security functions (presented in Chapter 4) in our GIZKA language.

A.3.1 Security functions for cryptographic support

Security functions with security parameters and security goals can be defined in our GIZKA language, as illustrated below where a character “\(\Rightarrow\)” means implication:
A.3.2 Security functions for user data protection

The class of security functions for user data protection can be illustrated in GIZKA as follows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15.</td>
<td>Known(X,P) =&gt; SecurityObjective.Availability;</td>
</tr>
<tr>
<td>16.</td>
<td>Combined(X,Y) =&gt; SecurityObjective.Integrity;</td>
</tr>
<tr>
<td>17.</td>
<td>Fresh(X) =&gt; SecurityObjective.Integrity;</td>
</tr>
<tr>
<td>21.</td>
<td>PartsOf(X,Y) =&gt; SecurityObjective.Integrity;</td>
</tr>
<tr>
<td>22.</td>
<td>Protected(X,P,Q) =&gt; SecurityObjective.Confidentiality;</td>
</tr>
<tr>
<td>24.</td>
<td>ObjectExchangedIntegrity(X,P,Q,X) =&gt; SecurityObjective.Integrity;</td>
</tr>
<tr>
<td>27.</td>
<td>InternalTransfer(X,P,Q) =&gt; SecurityObjective.Confidentiality;</td>
</tr>
<tr>
<td>29.</td>
<td>ResidualProtection(X) =&gt; SecurityObjective.Confidentiality;</td>
</tr>
<tr>
<td>30.</td>
<td>StorageIntegrity(X) =&gt; SecurityObjective.Integrity;</td>
</tr>
</tbody>
</table>

A.3.3 Security functions for identification and authorisation

GIZKA specifies these security functions as follows:
A.3.4 Security functions for security audit and resource utilisation

These security functions can be expressed in GIZKA in the following way:


A.3.5 Security functions for privacy

GIZKA defines security functions in the following way:

40. BelievesAnonymity(P,O,X) => SecurityObjective.Confidentiality;
42. Unobservability(P,O,X) => SecurityObjective.Confidentiality;

A.3.6 Security functions for trusted channels

The description of these security functions in GIZKA is illustrated as follows:

43. TrustedChannel(P,Q,M) => SecurityObjective.Confidentiality;
    TrustedChannel(P,Q,M) => SecurityObjective.Integrity;
44. TrustedPath(P,Q) => SecurityObjective.Confidentiality;
    TrustedPath(P,Q) => SecurityObjective.Integrity;
Appendix B

Introduction to security attacks and defences

In this appendix, we first describe different attacks against peer-to-peer (P2P) systems as well as countermeasures which enable to resist such attacks. Then, we present some of distributed denial of service (DDoS) and defenses against them. These security attacks and defences have been briefly introduced in Chapter 5.

B.1 P2P attacks and defences

B.1.1 Classes P2PInfrastructureAttack and P2PInfrastructureDefence

In the incorrect lookup routing attack (the class ‘P2PIncorrectLookupRoutingAttack’), a malicious node forwards lookups to an incorrect or non-existent node. For detection and prevention (the class ‘P2PIncorrectLookupRoutingDefence’) of such type of attacks, nodes should get closer to the key identifier at each hop lookup. Hence, a requestor must be allowed to observe a lookup progress. Also, it should use long term identities based on public-keys, so that system’s verifiability can be facilitated.

In the incorrect routing updates attack (the class ‘P2PIncorrectRoutingUpdateAttack’), a malicious node corrupts the routing table with incorrect updates to neighbours. Moreover, systems that have the freedom to choose between multiple routes are especially vulnerable to this attack. For resisting against such attacks, nodes need to use verifiable routing updates, based on the knowledge that the correct routing updates have certain requirements (the class ‘P2PIncorrectRoutingUpdateDefence’).

In the partitioning attack (the class ‘P2PPartitioningAttack’), a set of malicious nodes form a parallel network and trap new nodes inside them rendering the network useless for new nodes. For detecting such attacks users have to observe incorrect functioning of the network/queries etc. To withstand such attacks (the class ‘P2PPartitioningDefence’), there are three main mechanisms. One is that users use trusted source to bootstrap, and the source will likely be out-of-band to the system itself. Another one is that users should use history of queries and verify the current results in the network with random queries, so that the cross-check routing table can help users to detect such attack. Moreover, public-key have to be used to identify nodes in the
systems where a particular address is assigned via DHCP. For the partitioning attack, further actions should be taken to destroy the parallel network so to prevent new nodes being trapped.

**B.1.2 Classes P2PNetworkAttack and P2PNetworkDefence**

The pseudospoofing attack (the class 'P2PPseudospoofingAttack') exploits the way how P2P systems use pseudonyms. Pseudospoofing attackers create and control multiple phony identities. Such technique can be readily used to compromise P2P systems relying on pseudonyms rather than on full authentication. The pseudospoofing attack is relatively resilient to countermeasures based on peer reputations, as a malicious peer can use a new pseudonym after earning a bad reputation, and simulate multiple witnesses willing to give fake evidence in its favour. However, abandoning pseudonyms entirely would require a complex and expensive infrastructure capable of verifying the legal identities of all participants. The digest-based mechanism of approach makes pseudospoofing unprofitable, as doctored resources are recognized and discarded. Users have to check pseudonyms and IP addresses for protecting against pseudospoofing.

The shilling attack (the class ‘P2PShillingAttack’) is a well-known problem of distributed auctions protocols, where a malicious auctioneer may try to cheat his bidders by pushing up the price through creating multiple registration names (called skills) to bid under. This attack is a bit different from the pseudospoofing attack, as the multiple identities are actually created with the real IP addresses, and not just simulated, by the attacker. Since shilling and pseudospoofing are quite similar, then similar defence mechanisms can be used.

While achieving file sharing among users are provided in P2P systems, access control (the class 'AccessControl') must also be enforced to obey the copyright laws and protect the rights of certain users and companies. The accessibility of documents should be restricted only to those users that have paid for that access and have the right to access. It is still controversial whether or not it is reasonable to have the P2P network enforce access control, or whether it should be the endpoints of the network should take the responsibility of enforcing it. Some outcome can be found in [GD03,WN03]. Micropayment systems are becoming more mature in support of restricted access [MR02,KK99,YG03b].

Group-based authentication (the class ‘GroupBasedAuthentication’) is proposed in [GD03] where the model to enable the distributed authentication for P2P networks is described. In this model, whenever two peers want to establish a trust relationship, they form a group called troupe. Considering the dynamic property of P2P networks, a troupe formation protocol, a membership verification protocol and a troupe mutation protocol are provided. The core of the troupe formation protocol is RSA accumulator construction in a distributed way.
A third-party (the class ‘ThirdPartyInvolved’) involved access control is presented in [WN03], which enables a peer to transfer the right to access the encrypted data provided predetermined conditions are satisfied. The approach involves a third trusted service, called “delegation check servers” to check single or multiple conditions according to the regulations. A peer (delegator) delegates the right to decrypt the ciphertext to other peers (proxies) under certain conditions. The proxy can decrypt the ciphertext only after it passes the verification check of the delegation check server. In this way, access control has been achieved by encryption and verification.

In the micropayments model [MR02,YG03b] (the class ‘MicropaymentSystem’), peers offer something of value in exchange of opportunity to use peer-to-peer system resources. Peers should provide payment if they want to get some content or resources from system. It is done to prevent free-rider behaviours (peers that never allow access to their resources), and is considered as one kind of techniques to support access control. In the last few years, the micropayments have attracted quite a lot of publicity.

The class ‘FileAuthenticity’ defines file authenticity issues in P2P networks. Since P2P networks are more and more widely used in file sharing, and file authenticity is indispensable to ensure that resource consumers are getting the files they are searching for. Hence, invalid and bad content should be prevented from being retrieved by users. To further understand the problem of file authenticity, we would like to distinguish it from the problem of file integrity first. The goal of file integrity is to ensure that documents do not get corrupted due to communication failures. There are two main trends to solve the problem of file authenticity: to prevent and to differentiate, while the former is much more practical and popular. Proposal to prevent peers from distributing invalid and bad content into the network are mainly based reputation-based trust systems. Proposals to differentiate which file is authenticated are based on several options, such as oldest document (the class ‘OldestDocument’), expert-based (the class ‘ExpertBased’) and voting based (the class ‘VotingBased’) described in [DGY03].

In reputation systems (the class ‘ReputationBasedSystems’), one peer may interact with another peer only if it can trust to another peer. An electric reputation system is an effective way to facilitate the trust in a P2P system. It collects and aggregates the feedback of participants’ past behaviours and results are published in the network freely. Thus everyone can view it and determine who to trust in the future interactions. These models are based on system of voting, and there have been extensive studies on it.

### B.1.3 Classes P2PAnonymityAttack and P2PAnonymityDefence

**Attacks on anonymity**

The cache-timing attack (the class ‘P2PCacheTimingAttack’) aims to determine the contents of a user’s cache and his Internet usage patterns as well. By taking timing measurements, the
attacker can determine if the client had to fetch the requested content from the web or it was returned from the user’s cache. An alternative uses cache-cookies relying on similar principles. The timing attack (the class \texttt{P2PTimingAttack}) relies on the fact that web pages often contain multiple links that will cause the user to issue requests immediately upon receipt of the page. Knowing that, the attacker can uncover the initiator by observing sufficiently short delay between when he returns the response and when it receives the new request.

Assuming requests and responses of a unique session is identifiable, the predecessor attack (the class \texttt{P2PPredecessorAttack}) exploits the fact that any nodes in the network reside in any position of the communication path with equal probability except that the initiator occurs on every path in the first position. The attacker logs the nodes that forward requests to him. By applying the statistical analysis, the attacker notes the node which appears in his logs more often and is likely the initiator in a session.

Defences against attacks on anonymity

Server-based anonymization systems (the class \texttt{AnonymizationServerProxy}) often provided as commercial services on the web, ensure client identity is not directly exposed to the content provider in requests and content retrievals, by means of proxy server. A server-based anonymization system, however, constructs a potential single point of failure and performance bottleneck, and is susceptible to cache-timing attacks and timing attacks. Distributed anonymization systems, overcome the problem of single point failure, are more promising solutions (Onion Routing and Crowds [RR98]).

In Onion Routing (the class \texttt{OnionRouting}), an initiator determines the route for message flows and wraps the encrypted message recursively with public keyed encryption of routing information. Onion Routing is immune to traffic analysis and eavesdroppers. However, it cannot adapt rapidly to the changing structure of networks because all onion routers must be fully connected and any arrival or departure of them requires significant communication. Besides, the setup cost for each session is high due to encryption.

In Crowds [RR98] (the class \texttt{Crowds}), session paths are randomly formed and refreshed periodically. A peer joins the crowd by registering with a central server, performing a key exchange for encryption. When a peer makes a request, the initiator first sends its request to a random peer in the crowd. The intermediate peers make random decisions as to forward the request to another peer or to relay the request to the content provider. The response flows back to the initiator through the reverse path. The centralized blender is a bottleneck for performance, restricting the scalability of Crowds.

Anonymizing Peer-to-Peer Proxy (AP3) [WAL+01] (the class \texttt{AP3}) is an application layer proxy building on Pastry [RD01] that is modified from Crowds [RR98]. Communication in AP3
is sent in plain text, unlike in Crowds where messages are encrypted. Therefore, an eavesdropper can sniff packets and monitor all local communications. AP3 itself does not preserve sender anonymity or receiver anonymity; but with encryption in the Pastry layer, anonymity can be established. AP3 defends against cache-timing attack because AP3 nodes do not cache the content they request. On the contrary to Crowds, routes in AP3 are dynamic; the probability of choosing the same node in succession is low. Hence it is difficult for an attacker to perform timing attack. Predecessor attack is considered insignificant in AP3 because of the difficulty in uniquely identifying a session; and that with growing size of AP3 network, predecessor attack can hardly get sufficient collaborating nodes to succeed.

CliqueNet (the class ‘CliqueNet’) guarantees that an adversary cannot determine a request initiator beyond 3 to 5 hosts fully connected as a clique [SPR01]. It is organized as multiple cliques with overlapping nodes as ambassadors. Messages are encrypted and sent from clique to clique are broadcasted by ambassador to the destination clique, where the intended receiver picks them up by comparing a hash and decrypts them with its private key. CliqueNet guarantees sender anonymity because even if enormous computational power is given, receiver cannot determine the sender identity. CliqueNet is not susceptible to predecessor attack, because broadcast hides initiator identity in a crowd.

The above introduced protocols or schemes are designed for client / server paradigm, where sender anonymity is the main concern, client identity is hidden from the server with help of peer to peer networks. In peer to peer paradigm, where the content provider as well as the content requester are equally treated peers, not only initiator anonymity is required, but also responder anonymity. In other words, mutual anonymity is desired in peer to peer environment to protect user privacy where neither the information requester nor the provider can identify each other (Anonymous Peer-to-Peer File Sharing (APFS) [SLS01] (the class ‘APFS’), Peer-to-Peer Personal Privacy Protocol (P5) [SBS02] (the class ‘P5’) and Centre-Directing [XXZ03] approach (the class ‘CentreDirecting’)).

B.2 DDoS attacks and defences

B.2.1 DDoS attacks

The class BandwidthDepletionAttack

The bandwidth depletion attack (the class ‘BandwidthDepletionAttack’) floods the victim network with unwanted traffic that prevents legitimate traffic from reaching the primary victim.

The flood attack (the class ‘FloodingAttack’) involves zombies which send large volumes of traffic to a victim in order to overfill victim system’s network bandwidth with IP traffic.
Flooding attacks can be launched with the help of UDP (User Datagram Protocol) or ICMP (Internet Control Message Protocol) packets.

The UDP flood attack (the class ‘UDPAttack’) is one where the victim system receives a large number of UDP packets sent to its ports. It tries to process the incoming data to determine which applications have requested data. And then, if the victim system is not running any applications on the targeted port, it sends an ICMP packet to the sender indicating a “destination port unreachable” message. The source IP address also can be spoofed, i.e., return messages are sent back to the spoofed addresses.

During the ICMP flood attack (the class ‘ICMPAttack’) zombies send large volumes of ICMP_ECHO_REPLY packets (“ping”) to the victim system. These packets signalize the victim system to reply and network bandwidth is drained. The source IP address of ICMP packets might be spoofed either.

During amplification attacks (the class ‘AmplificationAttack’) the attacker (or zombies) sends messages to a broadcast IP address with a spoofed source IP address (victim’s address). It causes all systems in the subnet which receive these messages to send replies to the victim system, and thus reduce victim system’s bandwidth.

The class ResourceDepletionAttack

The resource depletion attack (the class ‘ResourceDepletionAttack’) relates to the situation in which the attacker sends packets that misuse network protocol communications in order to make the victim unable to process legitimate requests for services.

Subclasses of protocol exploit attacks (the class ‘ProtocolExploitAttack’) are the PUSH + ACK attack (the class ‘PUSHACKAttack’) where the attacker sends TCP packets with the PUSH and ACK bits set to one and the TCP SYN attack (the class ‘TCP SYN Attack’) that exploits the three-way handshake (the attacker sends TCP SYN requests with spoofed source IP addresses). Both of these attacks force the victim system to crash or be unable to provide services.

During malformed packet attacks (the class ‘MalformedPacketAttack’) the attacker instructs zombies to send incorrectly formed IP packets to the victim system in order to crash it. The subclass of the malformed packet attack is the IP address attack (the class ‘IPAddressAttack’) which meant that the packet contains the same source and destination IP addresses. It can confuse the operating system of the targeted victim system and, what is more, lead to its crash.

In the IP packet options attack (the class ‘IPPacketOptionsAttack’), a malformed packet may randomize the optional fields within an IP packet and set all quality of service bits to one in such way that the victim system has to use additional processing time to analyse network traffic. If this attack is multiplied, it may exhaust the processing ability of the victim system.
B.2.2 DDoS defence

The class DetectPreventSecondaryVictims
Individual users (the class ‘IndividualUsers’) and their systems should increase awareness of security issues and prevention techniques. For preventing secondary victims from being captured by DDoS agents, these systems must constantly monitor their own security (the class ‘BuiltInDefence’), install software patches (the class ‘InstallSoftwarePatch’), and signalize others when they participate in DDoS attacks.
Furthermore, network service providers (the class ‘NetworkServiceProviders’) can use dynamic pricing (the class ‘DynamicPricing’) charging users for the usage of the network in order to force secondary victims (i.e., users) to be more active in monitoring their own security. For example, if secondary victims pay for access to the Internet and usage of traffic, then they become more precise in policing themselves in order to verify if they do not participate in DDoS attacks.

The class DetectNeutraliseMasters
Another approach to resist DDoS attacks is to detect and neutralise masters, since shutting down a few masters, who usually control many agents, is more effective than removing only agents. To detect masters, the communication protocols and traffic patterns between masters and agents and between masters and clients should be studied.

The class DetectPreventPotentialAttacks
For detecting and preventing potential attacks, egress filtering (the class ‘EgressFiltering’) and MIB (Management Information Base) statistics (the class ‘MIBStatistics’) can be utilised which are both subclasses of the class ‘DetectPreventPotentialAttacks’.
Egress filtering is a method that scans IP packet headers leaving the network and verifies if they meet certain criteria (for example, an IP address of a packet is not spoofed). If packets satisfy criteria then they are allowed to be routed outside the network otherwise the packets are not sent.
The MIB statistics from routers allows detect ICMP, UDP, and TCP packet statistical abnormalities using statistical patterns, and consequently detect DDoS attacks.

The class MitigateStopAttacks
To mitigate and stop DDoS attacks (the class ‘MitigateStopAttacks’) three methods can be applied: load balancing (the class ‘LoadBalancing’), drop request (the class ‘DropRequest’), and throttling (the class ‘Throttling’).
Load balancing is the method when network providers increase bandwidth on critical connections in order to allow them to operate and do not go down and replicate servers. In our approach, we reconfigure the whole systems structure as well. Drop requests and throttling are two other methods proposed to mitigate DDoS attacks. The problem with implementing throttling is that it is hard to distinguish between legitimate traffic and malicious traffic.

**The class DeflectAttacks**

Honey pots (the class *Honeypot*), as well as Padded cells (the class *PaddedCell*), serve to deflect attacks (the class *DeflectAttacks*) from hitting the systems they are protecting as well as serving as a means for gaining information about attackers by storing a record of their activity and learning what types of attacks and software tools the attacker is using.

**The class PostAttackForensics**

The class *PostAttackForensics* consists of three subclasses including *TrafficPattern*, *EventLogs*, and *PacketTraceback*. For developing new filtering techniques, updating load balancing and throttling mechanisms, and improving countermeasures, data during DDoS attacks can be collected and analysed later. Such data is used for developing traffic patterns. Also, event logs and information from honey-pots and padded cells can be used to update traffic patterns. Another method called packet traceback is utilised to trace Internet traffic back to its original source.

**The class TCPSYNDefence**

The class *TCPSYNDefence* is used for describing techniques for resisting TCP SYN flood attacks that include the following:

- The system should increase the number of connections a server can support;
- The system should reduce the timeout period for waiting for the final ACK packet;
- The system should use the network-based IDSs to register such attacks;
- The system should use a “SYN cookie” that contains a special sequence number;

The system should use a “RST cookie” that is an alternative to “SYN cookie” and indicates the server that something goes wrong.
Appendix C

Security pattern ontology

In this appendix, we present and specify the security pattern ontology (SPO). SPO consists of two layers including the message layer and the system layer, as described in details in Chapter 6.

C.1 Message layer

Attackers may try to sniff the connections among components, hence, there is the need to secure such links. Security patterns for the message layer are represented by two classes ‘MessageInspector’ and ‘MessageInterceptorGateway’.

C.1.1 MessageInspector

This pattern inspects messages on all levels of the OSI models and is illustrated using GIZKA pseudo code in Figure C.1.

```plaintext
SECPATTERN MessageInspector {
[COMMENT]="This pattern verifies messages on all levels of the OSI models."
[EXAMPLE]="XML message-level security mechanisms (XML Signature and XML Encryption) as well as security mechanisms applied in SOAP messages can be checked."
//standards are specified in SASO
[STANDARDS]=[XMLEnc, XMLDSig, SAML, XKMS, AES, 3DES]
[RELATEDPATTERNS]=
[STATEGIES]=

//A message checking strategy.
STATEGIES CheckMessage {
//A message should be checked if it is encrypted.
    CheckEncryption(STRING Msg) {
        IF(Msg.ENCRYPTED) {
            RETURN TRUE
        } ELSE {
            RETURN FALSE
        }
    }
    CheckMsg(STRING Msg) {
        IF(CheckEncryption(Msg)) { // This function verifies if the message is encrypted or not
            IF(Decrypt(Msg)==TRUE) { // This function is responsible for decryption of messages
                IF(DetectAttack(Msg)==TRUE) { //This function enables attack detection
                    SEND ("A malicious encrypted message "+Msg) TO Msg.DESTINATION;
                } ELSE {
                
                }
            } ELSE {
                //normal message
                SEND (Msg) TO Msg.DESTINATION;
            }
        }
    }
}
```
C.1.2 MessageInterceptorGateway

This security pattern is employed as a single entry point in order to provide central security enforcement for all incoming and outgoing messages and is shown in Figure C.2.

C.2 System layer

Security patterns for the system layer are represented by several classes including ‘SingleAccessCheckPoint’, ‘PureP2PPattern’ and ‘SuperP2PPattern’, as presented below.

C.2.1 SingleAccessCheckPoint

This pattern enforces a single checkpoint of entry to the services and peers that provide a login prompt. It is illustrated in Figure C.3.
Figure C.3. SingleAccessCheckPoint

C.2.1 SecureCommunication

This pattern enforces encrypting traffic using specified algorithms such as HTTPS, SSL, TLS, and IPsec utilised for client-to-server, client-to-client (P2P), and server-to-server communications.
It is used to organise VPN connections between entities using various security technologies such as IPSec, cryptographic algorithms.

[STANDARDS] = {HTTPS, SSL, TLS, IPSec}

[STRATEGIES] = {
  // Strategy that specifies how to organise a secure connection between two components.
  STRATEGY CreateConnectionCC(ENTITY Comp1, ENTITY Comp2, Algorithm Alg) {
    PORT SecP1 = Comp1.WRAP(Comp1.PORT, Alg);
    PORT SecP2 = Comp1.WRAP(Comp2.PORT, Alg);
    SecP1 DYNAMIC BIND SecP2;
  }
}
Appendix D

A prototype of a social network system

In this appendix, we describe in more details the architecture of the prototype (presented in Chapter 8) of the social network system (SNS) (introduced in Chapter 7).

D.1 PROTO

D.1.1 Microkernel

As mentioned in Chapter 8, the ‘Core’ microkernel consists of three components including ‘Controller’, ‘Planner’, and ‘Queue’.

Controller

The ‘Controller’ coordinates the work of all other microkernel components executing such operations as starting and stopping the system, inserting and removing modules, showing debugging information, etc. Besides, the ‘Controller’ employs a number of PROTO internal commands:

- INSERT_MODULE <module> – inserts a module into the ‘Core’ relatively to other modules or by the absolute number;
- DELETE_MODULE <module> – deletes a module from the list of modules inserted into the ‘Core’;
- SHOW_MODULES – shows all modules inserted into the ‘Core’ and additional information about modules;
- START_MODULE <module> – starts a module;
- STOP_MODULE <module> – stops a module;
- SHOW_MESSAGES – allows to view the list of messages in the ‘Queue’;
- DELETE_MESSAGE <message> – deletes a message from the ’Queue’;
- HELP <keyword> – provides help information on the system and module usage;
- START_CORE_COMPONENT <core_component> – starts a core component;
- STOP_CORE_COMPONENT <core_component> – stops a core component;
- DEBUG_MODE <mode> – sets up a debugging mode. The debugging modes are:
General system modes – numbers from 1 to 20 are employed for general system modes such as:

- Runtime 1-3 – messages intended to debug procedures and functions;
- Receivable messages – messages added to and deleted from the ‘Queue’;
- Handle messages – messages that handle a status of each module;
- Global system states – messages that show changes of the global system state;
- Exceptions – messages that appear during handling of exceptions;
- Inactive states – messages that emulate operations of a module when it is inactive.

Reserved modes – numbers from 21 to 1023 are reserved for possible future needs.

- LOGIN <user> – allows a user to be identified and authenticated by providing user’s credentials such as login and password to perform some actions;
- LOGOUT <user> – allows users to leave the system;
- REGISTER <user> – registers a new user;
- CMD <command> – sends a command to a specific module and then this command should be executed by this module. If a command is successfully performed, it returns true and false otherwise;
- ENFORCE_SECURITY <security_command> – enforces security policies or mechanisms;
- RECALL_SECURITY <security_command> – voids certain security mechanisms or policies.

**Planner**

The ‘Planner’ supports an ordered list of modules connected to the ‘Core’, selects messages from the ‘Queue’ according to some criteria and transfers them to modules signed to work with this message type for further processing.

**Queue**

The ‘Queue’ contains messages with various system flags including type, priority, bandwidth, and so on in order to easily handle various types of data.

*(The general description)*

Every network module may place a new message into the ‘Queue’ by calling a method ‘putMessage’ provided by the message readout system (MRS). ‘putMessage’ forms the message itself by adding required headers. Then, MRS selects the next message packet to be handled and cyclically offers it to all modules assigned for handling this type of messages. A registered module for handling such messages analyses the message content and then deletes this message from the ‘Queue’ or leaves it unhandled. Each message has an attribute that specifies the
number of full inquiry cycles (a special counter is incremented each time after the whole cycle), which include sending of the message to all modules assigned for this message type. Then, such messages should be deleted from the 'Queue'.

Since modules are organized as an ordered list, they handle messages in the same order. For example, if a cryptographic protocol is used to encrypt a message, encryption and decryption modules can be inserted directly before and after the module that interprets the given message and make the interaction with a cryptosystem completely transparent to all other modules. The same approach is utilised if a large message is required to be sliced in smaller chunks and then placed onto the network and constructed on the recipient side from these small chunks.

(Message storage system)
Message storage system (MSS) contains a list of messages circulating through the 'Queue'. These messages have the following attributes:

- Security – contains a set of security fields algorithms used to protect a message itself and specify security requirements and capabilities of a sender or receiver;
- SecurityBit – a number indicating that the field 'Security' is used (1) or not (0);
- TimeToLive – a number of message inquiry cycles, after which (in case that there is no a debugger for a message) this message is deleted from the 'Queue';
- TimeLeftCounter – a number of inquiry cycles, which this message has already passed through;
- Type – a message type that is specified by a module which generated this packet and placed it into the 'Queue';
- MessageProtocolVer – a message version for providing compatibility since different versions of a module can generate the same message type;
- Priority – a message priority defines that messages with higher priorities should be handled prior to messages with lower priorities;
- Bandwidth – bandwidth is reserved for the given message type and supports gradation levels from 1 to 1000 (zero is reserved for messages for which bandwidth is not important);
- Message – a message contains specific information fields and a message body.

(Message readout system)
The message readout system (MRS) performs the readout of messages from the 'Queue' according to specific criteria and transfers them to plug-ins registered to handle this type of messages. The readout criteria are the following:

- A message with a certain priority is not handled unless there is one message remained with a higher priority in the 'Queue'. Active messages are all messages that present in the
‘Queue’ and have the active priority (the lowest priority of all messages in the ‘Queue’). Messages with priorities equal to the active priority are handled when there is a free time frame;

- Among the active messages, the one with a higher demand for the MRS in accordance with the value of the bandwidth field is to be handled. If several messages are handled simultaneously and it slows down the system, the MRS deletes certain messages for not disturbing functionality of other message flows and avoiding the possible bottleneck in the ‘Queue’. Besides, the MRS can assign weights in such priority scheme;

- Other active messages in the ‘Queue’ are handled according to the first-in-first-out principle. The list of message addresses in the readout system arranged in accordance with the bandwidth field with notification of the recommended time of message handling. At the same time, the ‘Queue’ is scanned to handle messages with the bandwidth field value equalled zero.

(Message types and formats)

Messages transferred through the ‘Queue’ consist of two parts:

- A header that defines external parameters important in the context of the system core;
- A body that may vary depending on the message type and that is formatted the same as the body of network messages, thus this approach provides higher efficiency in terms of message sending since no conversion of data is required.

To transfer data through the ‘Queue’, the same format as for the network protocol is used that considerably accelerates the transfer process. Thus, it is necessary to specify a set of message types and fields included into each message type. In accordance with the above mentioned module types, the following structure of types and fields is:

- ‘Sendable’ – messages assigned to be sent through a network.
  - Messages sent by using the UDP protocol.
    - Examples include audio and video streams with fields ‘To’ (to whom the message is addressed), ‘From’ (who sends the message), ‘StreamID’ (the stream number), and ‘Data’ (binary data in the stream).
  - Messages sent by using the TCP protocol.
    - Examples are large files with fields ‘To’, ‘From’ and ‘Data’ (binary data in the file); small chunks of large files with fields ‘To’, ‘From’, and ‘Data’ (message text); instant messages with fields ‘To’, ‘From’, ‘ReplyTo’ (where respond back), and ‘Data’ (message text); user status messages with fields ‘To’, ‘From’, ‘AvailValue’ (status field), ‘StatusDescr’ (current status), ‘MsgID’ (a message number); chat messages with fields
‘To’, ‘From’, ‘ReplyTo’ (where respond back), and ‘Data’; cryptographic messages with encrypted information with fields ‘To’, ‘From’, and ‘Data’.

- Messages sent using other protocols such as ICMP.
- ‘Receivable’ – messages received from a network.
  - This tree repeats the tree structure of ‘Sendable’ messages with an additional field such as the sender’s IP address. Modules can translate one type of messages to another and vice versa by substituting the sub-tree number. Thus, the translation is performed rather fast, and there is no disorder among sent and received messages.
- ‘ServiceState’ – messages for information exchange inside the system such as among modules regarding the system state changes with higher priorities or writing/reading to/from configuration files.
  - Examples include: a module registers the additional type to be handled within the ‘Queue’; a module finishes an operation; or the system connects a new module.

**Connection types**

PROTO allows several types of connections over the network:

- Client – Server: appears when a client connects to the system through a server;
- Client – Client (similar to P2P): appears when clients send messages to each other directly without using a server;
- Client – Server – System with another protocol: appears when a client connects to an external system through the use of gateway layers modules;
- Client – Super P2P: appears when a client connects to the system through one of super peers.

**D.1.2 PROTO modules**

Modules (plug-ins) are special components that deliver different functionalities and security features. New software applications with new features can be designed and developed by combining the microkernel with plug-ins (existed or newly developed), i.e., new features can be added to such software applications by adding new modules. There are three major types of plug-ins including user interface modules (UIL), processing modules (PRL), and gateway modules (GWL).

**UIL modules**

UIL modules are employed to interact with users (through a mouse, keyboard, microphone, etc.) and interpret and visualize packets from the ‘Queue’ and then encapsulate them back.
**PRL modules**

PRL modules interact with external processes (e.g., system events). They receive packets from the ‘Planner’, process them and then place the processed packets into the ‘Queue’. Examples of these modules are encryption and decryption modules used to secure messages; large files transfer modules that break large files into several smaller chunks; compression modules that employ compression algorithms to decrease the size of transferred messages; file sharing modules; storage and database modules; etc.

**GWL modules**

GWL modules are designed to listen to the network and place messages from different networks onto the ‘Queue’ as well as receive packets which have to be sent from the ‘Planner’ to the network. Each software application with such gateway can operate as a server/gateway to other networks (e.g., distributed counting networks, file sharing networks, chats, instant messaging networks, etc).

**Example modules**

As previously mentioned, modules are subdivided into three categories: UIL, PRL, and GWL. Examples of modules for the social network system (SNS) are illustrated in Figure D.1.

![Figure D.1. Example modules](image)

**Types of module connections**

The method used to plug-in modules to the ‘Core’ is static, i.e., it consists of static copies of each module and utilised to create a final version of the system.

**Module interfaces**

Every module should implement and satisfy the interface specified in PROTO which includes several functions:
• BOOLEAN insertModule(VOID) – is called, when a module is inserted into the ‘Core’. If initialization is successful, it returns true and false otherwise;

• BOOLEAN removeModule(VOID) – is called, when a module is removed from the ‘Core’. If removal is successful, it returns true and false otherwise;

• BOOLEAN handleMessage(MESSAGE) – handles a message and returns true, if this message is deleted from the ‘Queue’ after handling and false otherwise;

• STRING getInfo(VOID) – returns the string with information on the module purpose;

• PACKAGE[ARRAY] getAcceptedTypes() – returns the collection containing the copies of message types which are ready to be handled by a specific module;

• STRING getStatus(VOID) – returns the string with information on the current status of the module operation;

• VOID setDebugLayers(INT[ARRAY]) – sets the debugging layers for modules in order to deliver flexible testing;

• BOOLEAN cmd(MODULE,COMMAND) – sends commands to the module for an execution. If a command is successfully executed, it returns true and false otherwise.

Representations
We introduce the description of some modules and their representations in PROTO:

• secrobat.module.uil.video – a GUI interface that allows to watch stream video. In our prototype the Java Media Framework API (JMF) [Java06] that enables audio, video and other time-based media to be added to Java applications and applets;

• secrobat.module.prl.securityfilter – filters, enforces, recalls, or verifies security policies and properties of messages and modules as well;

• secrobat.module.prl.zip – allows reading and writing the standard ZIP and GZIP file formats as well as compressing and decompressing the data using the same algorithms;

• secrobat.module.prl.bigpacket – ensures that forwarding large files between two clients does not impede fast forwarding of short messages. It splits large files into several adaptive smaller chunks (depends on network bandwidth and system’s performance), marks them with lower priority, and places into the ‘Queue’. Then, these small fragments are transferred only when there is no need to forward messages with higher priorities;

• secrobat.module.prl.exec – executes simultaneously multiple PROTO commands incoming from the ‘Queue’ and supports their input/output;

• secrobat.module.prl.storage – stores all information on external sources such as hard drives or databases. For example, configuration data of all modules can be backed up by this module or this module can be employed to interact with the MySQL database [MySQL07]
responsible for storing Snort rules [Snort07] used by PROTO for detecting and analysing
security attacks;

- `secrobat.module.prl.msgbuffering` – supports two modes of information transfer including
direct messaging among online clients and through a server among online and offline clients
by buffering messages and forwarding them only when the addressee goes online;

- `secrobat.module.gwl.p2p` – allows peers to connect to each other directly. For example,
`secrobat.module.gwl.p2p.jxta` employs JXTA [JXTA04] in order to provide such
functionality;

- `secrobat.module.gwl.self` – is responsible for internal data transfers inside the system;

- `secrobat.module.gwl.self.ssl` – is responsible for delivering the same functionality as
`secrobat.module.gwl.self` but with the additional support of the SSL (Secure Socket Layer)
protocol which allows to organise secure connections. Data is encrypted before sending and
then decrypted on the recipients’ sides.

**D.1.3 Namespaces**

The addressing system for our SNS is specified as follows:

```text
<name>@<domain>[{!<hostN}>] where:

- `<name>` is a name which uniquely identifies the network participant (e.g., a client
  application or a system application) in the domain;

- `<domain>` is a domain name which uniquely identifies the group of the network participants
  unified by something common. To add a new participant to the domain, it is necessary to
  have a permission of the domain owner;

- `<hostN>` is a name of the host which is one of the system’s super peers (or servers). Since
  one participant can be connected through several servers or peers, her complete and definite
  addressing requires the notification through which server she should receive messages. The
  full message path should be stated as an enumeration of intermediate servers through the
  use of a symbol “!”]. The full path can be skipped if the system chooses the delivery path
  automatically.

It is necessary to mention that the ‘name’, the ‘domain’, and the ‘hostN’ can contain only
numbers, English letters, and symbols including ‘-’ and ‘_’. Every user or software agent has
the system address and the real IP address which uniquely distinguish the user.

**D.1.4 Development of peers**

In this section, we demonstrate how to develop the PROTO modules called ‘Chat’ and
‘Storage’ (the part of the client peers) for our social network system (introduced in Chapter 7)
using Java [Java06] and Groovy [Groovy07] programming languages. Actually, each of these
modules is a part of the client software applications that is used for chatting and videoconferencing by users of our social network system (SNS). Besides, the module ‘Storage’ is utilised to interact with the MySQL database [MySQL07] that stores Snort rules [Snort07] and used by the Basic Analysis and Security Engine (BASE) [BASE07]. In particular, BASE provides a web front-end to query and analyse alerts coming from a Snort system. In order to create pure P2P systems which can resist flooding DoS attacks, we employ JXTA [JXTA04] as a basis.

**Development of new PROTO modules**

To create a new software application (peer), a developer needs to implement new PROTO modules or utilise existing modules and connect them to the microkernel. Each implemented module is a Java or Groovy class that realises two compulsory methods including ‘getAcceptedTypes()’ and ‘handleMessage()’ and the needed functionality. More specifically, every PROTO module should register its message types using the method ‘getAcceptedTypes()’ to allow the microkernel to process these messages. Besides, every module should implement the method ‘handleMessage()’ used to handle the registered messages (following prescribed logic). As previously mentioned, messages are employed to organise external and internal communications among various system’s components. The example of the Java source code of both methods is illustrated in Figure D.2. In this example, the method ‘getAcceptedTypes()’ registers two message types such as ‘StorageTCPReceivableMessage’ and ‘ChatTCPReceivableMessage’ which are used to transfer data from/to the ‘Storage’ and ‘Chat’ module accordingly.

```java
public boolean handleMessage(MessageInterface msg) {
    //handling messages using prescribed logic
    .............
    return false;
}
public Message[] getAcceptedTypes() {
    Message a[] = {
        /*registered messages*/
        StorageTCPReceivableMessage,
        ChatTCPReceivableMessage
    };
    return a;
}
```

*Figure D.2. Methods ‘handleMessage()’ and ‘getAcceptedTypes()’*

After the message has been registered, it should be implemented. The short example of the message ‘StorageTCPSendableMessage’ implemented in Java and supported by the module ‘Storage’ is shown in Figure D.3. It includes compulsory fields such as ‘Module’ (a module that
generated a message), ‘From’ (who sent a message), ‘To’ (a message was sent to whom), and ‘ReplyTo’ (who should get a reply). For more simplicity, we avoid showing underlying logic of this method.

```java
public class StorageTCPSendableMessage extends TCPSendableMessage {
    private byte[] StorageMessage;
    public StorageTCPSendableMessage() {
    }
    public StorageTCPSendableMessage(
            GenericModuleInterface Module,
            String From,
            String[] To,
            String[] ReplyTo,
            byte[] message) {
        super(Module, From, To, ReplyTo);
    }
    //underlying logic
}
```

**Figure D.3. The class ‘StorageTCPSendableMessage’**

After all compulsory methods and required logic for the certain module (in our case the module ‘Storage’) has been implemented, we can plug it into the microkernel. This process is illustrated in Figure D.4 where: (1) the microkernel is launched; (2) All types of messages are registered; (3) the implemented modules are loaded.

```java
class StorageServer {
    static private Queue m_Queue;
    public void init() {
            m_Queue = new Queue(this)).start();
    }
    public static void main(java.lang.String[] args) {
        Queue m_Queue = new Queue(null);
        //start the microkernel
        m_Queue.start();
        //register all types of messages
        Util.registerTypes(m_Queue);
        //load modules
        m_Queue.loadModuleFromClass(new secrrobat.module.prl.storage,0);
        ..................
}
```

**Figure D.4. The class ‘StorageServer’**

**Development tools**

In the previous section, we briefly have explained how to implement a new module in Java. In this section, we demonstrate how to use the Eclipse Software Development Kit (SDK) [Eclipse07] to develop new modules in Java and Groovy. More specifically, we show an example of the class ‘Database’ implemented in Java that is the part of implementation of the module ‘Storage’ responsible for storing data in files or databases locally or remotely. The class ‘Database’ is capable to store and retrieve data generated by the module ‘Chat’ that utilises the module ‘Storage’ that connects to the MySQL database [MySQL07].
Besides, the class ‘Database’ is utilised to read/write Snort rules (used for attack identification and analysis) from/to the MySQL database [MySQL07]. The screenshot of the Eclipse SDK is shown in Figure D.5. The piece of the source code of the class ‘Database’, which is the part of the Java package ‘secrobat.module.prl.storage’, generated by the Eclipse SDK is illustrated in Figure D.6.

```java
package secrobat.module.prl.storage;
import java.sql.*;
import java.util.*;
import secrobat.debug.*;

public class Database {
    private DatabaseAccessor dbAccess;
    public Database() {
        dbAccess = new DatabaseAccessor();
        dbAccess.connect();
    }
    public Database(String driver, String host) {
        dbAccess = new DatabaseAccessor(driver, host);
        dbAccess.connect();
    }
    public Database(String driver, String host, String username, String password) {
        dbAccess = new DatabaseAccessor(driver, host, username, password);
        setDatabaseUser(username, password);
        dbAccess.connect();
    }
    public void close() { dbAccess.disconnect(); }
    public boolean createQuery(String queryStr) {
        // Code for creating query
    }
}
```
if(dbAccess.queryDB(queryStr)==null || !dbAccess.closeQuery()) {return false;} else {return true;}
}
public Store findStoredObjectByID(int ID) {
    ResultSet rs = dbAccess.queryDB("SELECT * FROM Store WHERE ID="+ID);
    if (rs == null) {
        Trace.debug("Result set null in findStoredObjectByID(ID="+ID+")");
        return null;
    }
    Store store = null;
    try {
        if (rs.getInt("ID") == 0) { return null; }
        store = new Store();
        store.setID(rs.getInt("ID"));
        store.setName(rs.getString("Name"));
        store.setObject(rs.getBlob("Object"));
    } catch (SQLException sqle) {
        Trace.debug("SQL Exception caught in findStoredObjectByID(ID="+ID+")");
    }
    return store;
}
public Vector findByMask(String mask) {…}
public Store findStoredObjectByName(String name) {…}
public void insertStoredObject(Store store) {…}
public void removeAllStore() {…}
public void removeAllStore(Store store) {…}
public void removeStoreByID(Store store) {…}
public void removeStoreByName(Store store) {…}
public void setDatabaseUser(String username, String password) {…}
public void updateStoreByID(Store store) {…}
public void updateStoreByName(Store store) {…}
}

Figure D.6. The class Database

As shown in Figure D.6, we import the required packages for working with the MySQL database (DB) ('java.sql.*'), vectors ('java.util.*'), and debugs ('seerobat.debug.*'). After this, we declare constructors that initiate a connection with the DB using the class 'DatabaseAccessor' and its method 'connect()'. Besides, the user can log in to this DB anonymously or using 'username' and 'password'. Another method called 'close()' uses the method 'disconnect()' of the class 'DatabaseAccessor' to close the connection with the DB. Other methods of this class are utilised to execute SQL queries on the DB including create queries (the method 'createQuery()'), find data in the DB using a specified mask (the method 'findByMask()'), find a serialised object (e.g., a message) stored in the DB (the methods 'findStoredObjectByName' and 'findStoredObjectByID'), insert serialised objects to the DB (the method 'insertStoredObject'), remove data from the DB (the methods 'removeAllStore', 'removeStoreByID', and 'removeStoreByName'), and update data in the DB (the methods 'updateStoreByID' and 'updateStoreByName').

Application scenario
In this section, we demonstrate how a chatting software application (peer) can be used for sending text messages between users (clients) of our SNS. It is necessary to mention that this application employs the ‘Chat’ and ‘Storage’ modules and its class ‘Database’ to write/read data (related to chat-messages and service messages) to/from the MySQL DB.

The process of chatting can be illustrated in the following way. For example, there are two clients (client1 and client2) who want to chat using our SNS. After both clients have logged in, they can start talking. Their short dialog is illustrated in Figure D.7.

client2<< Hello, client1. How are you?
client1<< Hello, client2. I'm fine, thanks. And you?
client1<< Bye.

Figure D.7. The dialog between two clients

The GUI part of the ‘Chat’ module is implemented in Java Swing and shown in Figures D.8-D.9. First, client2 says to client1 “Hello, client1. How are you?”, as illustrated in Figure D.8 where client1 is on top of the presented below figures while client2 is on bottom of each figure.

Figure D.8. The dialog (1)
Then, \texttt{client1} answers “\textit{Hello, client2. I'm fine, thanks. And you?}”. Finally, \texttt{client2} says “\textit{Fine too. See you soon. Bye.}” and finishes this dialog. The resulting dialog is shown in Figure D.9.

\section*{D.2 Internal architecture of peers}

In this section, we describe the internal architecture of peers, as introduced in Chapter 8.

\subsection*{D.2.1 Defensive peers}

The internal architecture of the defensive peers has been presented in Chapter 8. In this section, we describe in more details the software packages used to implement defensive peers.

\textbf{Apache Web Server}

Apache Web Server [Apache07] has been one of the most popular web servers on the Internet since 1996. In our case, Apache is solely employed to host the BASE web-based security console [BASE07], PHP [PHP07] and MySQL [MySQL07].

\textbf{PHP}

PHP [PHP07] is an open source server-side scripting language that is especially suited for Web development and can be embedded into HTML [HTML07] (similar to JSP [JSP07] which requires Apache Tomcat [Tomcat07]). PHP derives its syntax from C/C++ and Perl and it is easy to learn and apply.
MySQL
MySQL [MySQL07] is an open source SQL based database server, for both Unix and Windows based systems and is the supported platform for storing Snort alerts triggered from Snort sensors. The combination of PHP and MySQL has become a very common environment to develop fast, powerful and secure web applications.

ADODB
ADODB [ADODB07] is an object oriented library developed in PHP for abstracting database operations for portability.

Snort
Snort [Snort07] is an open-source network intrusion detection system that is capable to perform packet logging and real-time traffic analysis on IP networks. This is the software package that is used to gather information from the network.

WinPcap
WinPcap [WinPcap07] is a software tool that allows capturing and sending raw data from a network card as well as filtering and storing the captured packets in a buffer.

BASE
The Basic Analysis and Security Engine (BASE) [BASE07] is a web-based application for viewing the Snort IDS alerts from Snort sensors. For example, BASE allows to receive additional information about attacks from various sources including snort.org.

SnortSam
SnortSam [SnortSam07] is an open-source plug-in for Snort that allows automated blocking of IP addresses on the several firewalls including Checkpoint Firewall-1, Cisco PIX firewalls, Cisco Routers, Juniper firewalls, IP Filter (ipf), FreeBSD's ipfw2, OpenBSD's Packet Filter (pf), Linux IPchains, Linux IPtables, Linux EBtables, WatchGuard Firebox firewalls, 8signs firewalls for Windows, MS ISA Server firewall/proxy for Windows, CHX packet filter, etc. SnortSam consists of two parts including the output plug-in within Snort and an intelligent agent that runs on the firewall.

8signs firewall
Since we deploy our Snort sensors on Windows based systems and we want to utilise SnortSam, thus, we employ 8Signs Firewall [8Signs07] as a part of our reconfiguration process because it
can block attackers. 8Signs Firewall is a stateful inspection firewall that determines whether packets can get through the firewall based on the protocol, port, and source and destination addresses. Besides, it allows trap hackers by creating tarpits which is a trap for malicious users. Usually, a tarpit accepts TCP connections but never replies and ignores disconnect requests in order to leave ports scanners and hackers stuck a long period of time. Also, 8Signs Firewall allows to ban IP addresses to prevent unwanted connection attempts, filter by MAC address, protect against TCP SYN attacks etc.

Operational process
After installing all software packages and modifying configuration files, we rewrite source code of BASE and Snort for our needs and create MySQL database called snort which is used for storing Snort alerts. Besides, we register Apache Web server, MySQL database server, and Snort as Windows services. Then, we launch the 8Signs firewall and SnortSam to allow Snort to identify and block attacks. The detailed instruction is not presented here due to the complexity and dependability on a selected host operational system, a firewall, and an IDS. Finally, the manager (described in Chapter 8) can see Snort alerts and manage Snort using BASE and the specific PROTO modules and decide if an attack is happening or not, as illustrated in Figures D.10 and D.11.

Figure D.10. Alert processing
Figure D.11. Snort alerts