A4.D1.1 - Review of the state of the art

G. Spanoudakis, C. Kloukinas, T. Tsigritis, K. Androutsopoulos, C. Ballas and D. Presenza

Document Number	A4.D1.1
Document Title	Review of the state of the art (in Security and Dependability Monitoring and Recovery)
Version	1.0
Status	Final
Work Package	WP 4.1
Deliverable Type	Report
Contractual Date of Delivery	31/March/2006
Actual Date of Delivery	4 April 2006
Responsible Unit	CUL
Contributors	CUL and ENG
Keyword List	Dynamic verification of system security and dependability
Dissemination level	PU

Change History

Version	Date	Status	Author (Unit)	Description
0.1	20 February 2006	Draft	C.Kloukinas (CUL) G.Spanoudakis (CUL)	Draft table of contents.
0.2	15 March 2006	Draft	Theoharis Tsigritis, Kelly Androutsopoulos, Costas Ballas, George Spanoudakis, Christos Kloukinas (CUL)	Initial draft.
0.3	24 March 2006	Draft	Domenico Presenza (ENG)	Addition of immunisation base monitoring approaches.
0.4	31 March 2006	Draft	Theoharis Tsigritis, Kelly Androutsopoulos, Costas Ballas, George Spanoudakis, Christos Kloukinas (CUL)	Integrated draft.
1.0	4 April 2006	Final	George Spanoudakis, Christos Kloukinas, Theoharis Tsigritis, Kelly Androutsopoulos, Costas Ballas (CUL)	Deliverable circulated to consortium.

Executive Summary

This deliverable provides a review the state of the art in security and dependability monitoring and recovery in order to identify approaches and techniques that have been developed for these tasks and analyse their strengths and limitations.

Our review has identified the basic architecture of runtime monitoring systems, the common features of the languages that may be used to formalise security and dependability properties that need to be verified at runtime, and the main issues in connection with dynamic verification which are open to further research.

The identified research issues are related to: (i) the need to provide support for transforming of specifications of security and dependability properties that need to be monitored at runtime into the event patterns that should be observed to verify them, (ii) the need to develop mechanisms that can support the diagnosis of the reasons underpinning run-time violations of security and dependability properties requirements that could inform system adaptation to ensure that violations will not re-occur, (iii) the ability to support the specification of end-user personal and ephemeral security and dependability properties, the automatic assessment of whether or not such properties can be monitored at run-time, and the transformation of these properties onto monitorable patterns of run-time events, (iv) the development of techniques that would allow the identification of scenarios of potential security and dependability threats (i.e. potential violations of security and dependability properties that have not occurred yet but seem to present a realistic possibility for the subsequent operations of a system) and the translation of these scenarios into monitorable event patterns that would allow the development of pro-active techniques for protecting security, (v) the need to develop mechanisms that can ensure that the events used in dynamic verification have not been altered by an attacker in order to affect the results of the verification process and consequently the recovery actions that may be taken in response to these results, and (vi) the need to develop monitors that could detect violations of security and dependability properties efficiently and timely so as to allow the effective reaction to such violations.

The above issues establish a roadmap that will inform the research in Activity 4 of the SERENITY project.

Table of Contents

1. Introduction

1.1. Scope and Objectives

The objective of this deliverable is to provide a review the state of the art in security and dependability monitoring and recovery in order to identify approaches and techniques that have been developed for these tasks and analyse their strengths and limitations. To this end, it aims to establish a basic roadmap for the research in Activity 4 of the SERENITY project which is concerned with the development of a framework to support: (a) the dynamic (i.e., runtime) monitoring and verification of security and dependability requirements and solutions for interoperable ambient intelligence ecosystems, (b) the diagnosis of violations of and threats to these requirements and solutions, and (c) the timely recovery from such violations and reaction to threats when they occur.

Avizienis et al. [21] have defined dependability as "the ability of a (computer) system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)" and "deliver service that can be justifiably trusted". The notion of service in this definition corresponds to the system behaviour as viewed by the user, who may be a human interacting with the system or another system. A service delivery is acceptable if it implements the required system behaviour and satisfies certain quality constraints while failures relate to events that make the service deviate from what is perceived to be a correct delivery.

An important element in the above definition of dependability is the notion of "justifiable trust" which requires the ability to objectively verify that the delivered system service does not deviate from the required system behaviour and associated quality constraints. The development of system verification capabilities (i.e., the ability to verify that a system satisfies certain properties) has been the focus of significant research over the last few decades and has resulted in the development of a wide spectrum of, typically tool-supported, methods that offer such capabilities. These methods are distinguished into *static* and *dynamic*.

Static verification methods aim to show that the desired properties of a system will always hold based solely on the specification of the system without considering its actual run-time behaviour. Dynamic verification methods, on the other hand, aim to show that desired properties hold based on observation of the run-time behaviour of a system and its interactions with its operational environment.

Whilst static verification is not the main area of interest of this survey, we provide a brief overview of methods that fall in this category in order to demonstrate their main similarities and differences from dynamic verification methods, and demonstrate weaknesses that make it necessary to deploy dynamic verification. Our overview of static verification methods focuses on the use of formal methods for developing and analysing security systems and their properties. We also focus on the formal verification of cryptographic protocols as a significant amount of research has been devoted to this area with significant accomplishments. The main issues that we have considered in connection with the analysis of cryptographic systems include: the modelling (if any) of intruders that aim to compromise system security and dependability, the specification of security properties including the degree of its formality, and the extent to which verification is automated.

Dynamic system verification has emerged more recently and has been investigated in the context of different areas including requirements engineering, program verification, safety critical systems and service centric systems.

In requirements engineering, dynamic verification has focused on system requirements and investigated: (i) ways of specifying requirements for monitoring and transforming them into events that can be monitored at run-time; (ii) the development of event-monitoring mechanisms; (iii) the development of mechanisms for generating system events that can be used in monitoring (e.g. instrumentation [204], use of reflection [53] ; and (iv) the development of mechanisms for adapting systems in order to deal with deviations from requirements at run-time as, for example, in [238].

In dynamic program verification, existing work has focused on the development of programming platforms with generic program monitoring capabilities including support for generating program events at run-time (e.g. jMonitor [134], embedding specifications of monitorable properties into programs and producing code that can verify these properties during the execution of the programs (e.g. monitoring-oriented programming [59]). A significant body of this research has been published in the proceedings of the series of Workshops on Runtime Verification[1] that started in 2001.

In safety critical systems, dynamic verification methods emerged in order to provide more formal system verification than testing [12]. Early work in this area focused on run-time monitoring of embedded and safety-critical real-time systems in order to detect timing failures and guarantee system responsiveness [129, 183, 234]. Later run-time monitoring techniques were applied to autonomous safety critical systems such as NASA's autonomous Deep Space Remote Agent [193], as the testing of such systems was difficult and very resource consuming due to their high complexity systems. Moreover, runtime monitoring was seen as a mechanism enhancing the autonomy of such systems.

In service-centric systems – i.e. systems which "are implemented from autonomous web services coordinated by some composition process" [166] – the interest in dynamic verification has emerged due to the need to be able to specify and monitor *service level agreements* between the providers and consumers of web-services which are deployed in service centric systems. As a result of recognizing the importance of this form of verification, work in this area has focused on the development of standards and languages for specifying monitorable service level agreements (e.g. *WS-Agreement* [15], *WSLA* [164]) and methods for monitoring them [26, 166] .

Research on dynamic verification has also focused on system security. Work in this area has mainly been concerned with the development of *Intrusion Detection Systems (IDS)* which use dynamic verification techniques for detecting security threats. Our review covers this area along with classic approaches for monitoring and supporting security, such as firewalls.

An increasingly important requirement for system security and dependability is the ability to recover from attacks or system faults identified at run-time, and possibly adapt to handle these attacks and faults. Effective recovery requires the provision of diagnostic information of the nature and cause of the attack or fault, and the development of flexible system architectures and system deployment environments that can undertake recovery actions at run-time. Despite the recognition of its importance, there is little work that focuses on recovery. Our review of the literature has identified that most of the work on recovery arises from the areas of safety-critical systems (focusing mainly on fault-tolerance) and databases. Consequently, our survey focuses on these areas and discusses recovery for safety-critical systems (aimed mainly at preserving dependability),

[1] http://react.cs.uni-sb.de/rv2005/

mission-critical distributed systems (aimed mainly at preserving survivability) and trusted recovery as part of information warfare defence.

Run-time monitoring is particularly well suited to detecting known hazardous conditions while detecting unknown or unexpected hazards is lot more challenging [165].

1.2. Document Structure

The rest of this document is structured as follows:

Section 2 provides an overview of the main types of security requirements that have been identified in the literature and a special type of such requirements of an increasingly emerging importance which are related to digital rights management.

Section 3 reviews the state-of-the-art in the static verification of security properties for achieving dependability.

Section 4 reviews the state-of-the-art in the dynamic verification of security properties for achieving dependability.

Section 5 reviews research related to system recovery following breaches of dependability and other system properties. Our review has focused mainly on recovery methods used in the areas of safety-critical systems, mission-critical distributed systems and trusted recovery of information warfare defences, as pure security oriented recovery is an emerging area with relatively less research devoted to it.

Finally in Section 6, we present the main open research issues regarding dynamic verification and recovery that were identified by our survey and conclude this survey.

2. Security Requirements

2.1. General Security Requirements

Security requirements cover issues related to [228]:

— Confidentiality – Confidentiality is the ability to maintain the secrecy of stored system data and the messages exchanged by a system and its collaborating actors over networks.

— Integrity – Integrity is the ability to ensure the accuracy and completeness of the data stored and the messages exchanged by a system. Maintaining integrity involves allowing only authorised users to change or create data and messages and applying controls to ensure the correctness of these messages and data.

— Availability – Availability is concerned with ensuring that access to a system is possible when required.

— Non-repudiation – Non- repudiation is concerned with making it impossible for an entity that has participated in some communication with a system to deny this participation. In message exchange, for instance, non- repudiation guarantees that the sender and the receiver of a message cannot deny the dispatch and receipt of the message, respectively.

— Authentication – Authentication is the ability to determine whether an actor interacting with the system has the identity that it claims to have.

— Authorization – Authorization is concerned with the assignment of the right permissions to an already authenticated entity.

— Privacy – Privacy is the ability of a system to prevent personal information from becoming known to entities other than those which own the information or the information is about.

2.2. Requirements related to digital Requirements related to Digital Rights Management

Another important family of security requirements and properties is the one dealing with what is known as Digital Rights Management (DRM) requirements, which will allow us to obtain a different perspective on the usual security properties and solutions. The goal of Digital Rights Management (DRM) is to protect the rights of owners of digital content, by specifying which kinds of uses are acceptable and which are not. At a high level DRM deals with the control of the rights to the digital content (e.g. how it can be used, how often, etc). DRM is applicable to a variety of types of content (e.g. music, books, video). Different definitions for DRM have been proposed but the European Standards Committee has proposed the following one [56]:

"Digital Rights Management(DRM) is the management of rights to digital goods and content, including its confinement to authorized use and users and the management of any consequences of that use throughout the entire life cycle of the content."

Rosenblatt et al. [208] have also provided two definitions for DRM. With their first definition they refers to the technology that protects the digital content via encryption and access control mechanisms ("*DRM is a persistent protection of digital data*"), while with the second one they refer to technology that must be used in order to manage and track digital content on the Internet ("*DRM is everything that can be done to define, manage, and track rights to digital contents*").

The broader use of the term DRM includes the description, identification, trading, protection, monitoring and tracking of all forms of rights usages over tangible and intangible assets including management of the rights holders relationships.

Kurth [151] has pointed some of the advantages of a wide spread use of a DRM framework, such as the protection of the rights of the owners of digital works but also the fact that DRM can enable micropayment schemes, where users only pay for what they consume or are billed on a "per use" basis. Also micropayments in conjunction with digital distribution can lower or even essentially eliminate transaction costs. Since the cost of digital distribution is zero content creators can distribute their work immediately after production. This can reduce prices for consumers while content creators can market their creations without having to work with a publishing company.

Matheus [171] identified the following functions in order to guarantee a working DRM system:

- Authentication is the process of determining whether someone or something is, in fact, who or what it is declared to be. In private and public computer networks (including the Internet), authentication is commonly done through the use of something secret (e.g. password, PIN), something you have in possession (e.g. smart card) or something that you are (e.g. finger print, iris scan). In a DRM system authentication specifies the means for proving identities of the provider of digital content, the licensor, the licensee as well as proofing the authenticity of a digital content and other entities like services and systems.
- Access control is the ability to permit or deny the use of an object (a passive entity, such as a system or file) by a subject (an active entity, such as an individual or process). In a DRM system the decision is based on a concrete request and a set of formalized rights. This set of rights contains not only access rights but important for the DRM system rights about copying, distribution and loan.
- Digital Signatures and Encryption defines the means for establishing confidentiality, integrity and authenticity of the digital content and its communication. In particular encryption can be used for the enforcement of licensed use for a digital content. Digital Signatures can provide the means for proofing authenticity of digital content, legal entities like users as well as services or systems. Also watermarking and fingerprinting can be used, under certain circumstances, for proofing the authenticity of digital content and the legitimate owner of digital content. In any case, these techniques can provide the means to ensure that the content can only be used by the legitimate owner under certain rules.
- Delegation of Rights or Licensing is distinctively difference than access control. With access control, the access rights are typically enforced on the object side, where an access control decision is taken based on a concrete request and a set of formalized access rights. With licensing, rights like copying, total number of playback, etc. are also meaningful. These rights must be enforced on the user's system to support offline use. Therefore, a license contains a set of permissions, which express the rights of the subject, identifying the licensor and licensee.

2.2.1. Distribution architectures of DRM systems

Park et al. [199] distinguished different possible distribution architectures that could be implemented for DRM enabled data distribution, based on the following three factors: the presence of a virtual machine (VM), the type of control sets and the distribution style.

- Virtual Machine: The Virtual Machine (VM) is described by Park et al. [199] as "software that runs on top of vulnerable computing environment and employs control functions to provide the means to protect and manage access and usage of digital information". A VM can be in the form of a plug-in that controls access to DRM enabled data or embedded in the application itself. Systems that do not have a VM cannot manage and control the access and usage of DRM enabled data.
- Type of Control Sets: Control sets are the rules governing the use of DRM enabled data, using Right Expression Languages (REL) which allow the description and specification of the control sets. Control sets are distinguished into three types: fixed control sets, embedded control sets and external control sets. In fixed control sets, the virtual machine comes with a predefined control set which is enforced for all DRM enabled data. In an embedded control set, the DRM enabled data comes with the control set embedded into the work. This can be done by encapsulating the control set and the data in a security envelope. Finally, in an external control set, the DRM enabled data and the control sets are being distributed separately. The obvious advantage of this type of control set is that a single control set can be used to define rights for multiple works of the same type. A fixed control set can be combined with either an embedded or an external control set.
- Distribution process: The final distinction is in the distribution process. Park et al. [199] proposed two types of distribution: message push and external repository. In a message push system, the data is transferred from the owner to the buyer through a direct communication channel such as e-mail. In the case of an external repository, the buyer fetches the data from a central repository and there is no need to store them locally (e.g., a RealAudio feed of a radio station programme). Both systems have their uses in DRM systems and the choice of distribution system does not necessarily impact on the security of the data. Message push systems are useful in enterprises where the data is only meant to be available to specific employees. Message push also has a greater flexibility in managing individual right permissions. External repositories are useful for a wider range of deployment, where the prospective user is unknown. They can also be used in systems where the user cannot store the data permanently onto their own systems. This type of DRM allows the right holder a high degree of control in how the user accesses and uses the data.

Based on these factors Park et al. [199] identified eight different architectures for a DRM system. Figure 1.1 illustrates these architectures.

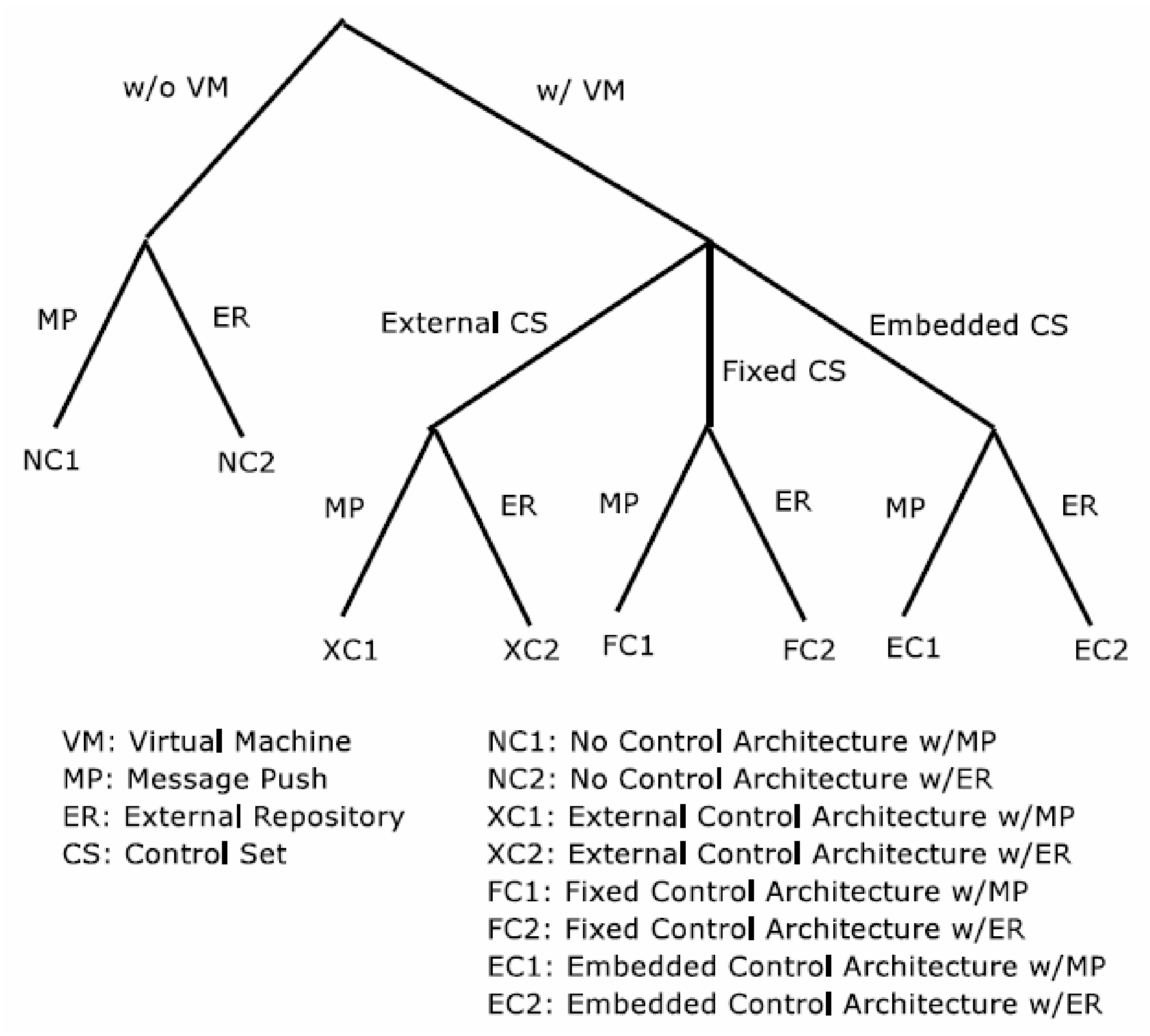

w/o VM
w/ VM
MP
ER
External CS
Fixed CS
Embedded CS
NC1
NC2
MP
ER
MP
ER
MP
ER
XC1
XC2
FC1
FC2
EC1
EC2
VM: Virtual Machine
MP: Message Push
ER: External Repository
CS: Control Set
NC1: No Control Architecture w/MP
NC2: No Control Architecture w/ER
XC1: External Control Architecture w/MP
XC2: External Control Architecture w/ER
FC1: Fixed Control Architecture w/MP
FC2: Fixed Control Architecture w/ER
EC1: Embedded Control Architecture w/MP
EC2: Embedded Control Architecture w/ER

Figure 1.1 – Distribution architectures of DRM systems – (Park et al. 2000)

2.2.2. Rights Expression Languages

Rights expression languages (RELs) are used to define the rights and conditions for DRM enabled data that the rights holder gives to the user. RELs are usually modelled on access control languages, and usually take the form of: <USER> has the <RIGHT> to do an <ACTION> on the DRM enabled data. This can be enhanced by including parameters that restrict the right.

The two most common RELs are the eXtended Rights Markup Language (XrML) and the Open Digital Rights Language (ODRL). The XrML [251] was developed originally at Xerox Parc labs and is now developed jointly by Microsoft and Xerox. XrML is an XML based REL, and its syntax is specified in XML. The XrML 2.0 specifications are split into three parts: a core schema, a standard extension schema to handle definitions that are broadly applicable but not a core feature, and a content specific extension schema to handle concepts specific to the type of digital content.

The XrML data model consists of four entities and the relationships between these entities. The basic relationship is defined by the XrML assertion "grant". Structurally, an XrML grant consists of the following:

— The principal to whom the grant is issued

— The right that the grant specifies

— The resource for which the right is granted

— The condition that must be met for the right to be exercised

So, in an XrML license, the issuer grants a set of principals with a set of rights under certain conditions for a set of resources. While the resources are usually digital files, XrML also provides mechanisms to include non digital objects such as "a computer terminal", as well as, services and transactions.

In the following example we can see the four entities of the XrML data model and the relationships between them. The following license consists of:

— `keyHolder` who is the principal designated with an RSA key.

— `print` which is the right being granted.

— `digitalWork` which is the resource specified as URI.

— `validityInterval` which is a condition permitting usage until Christmas 2001.

```
<license>
    <grant>
          <keyHolder>
               <info>
               <dsig:KeyValue>
                    <dsig:RSAKeyValue>
                         <dsig:Modulus>Fa7wo6NYfmvGqy4ACSWcNmuQfbe
                              jSZx7aCibIgkYswUeTCrmS0h27GJrA15SS7T
                              YZzSfaS0xR9lZdUEF0ThO4w==
                         </dsig:Modulus>
                         <dsig:Exponent>AQABAA==</dsig:Exponent>
                    </dsig:RSAKeyValue>
               </dsig:KeyValue>
               </info>
          </keyHolder>
          <cx:print/>
          <cx:digitalWork>
               <cx:locator>
```

```
                    <nonSecureIndirect
                         URI="http://www.sellbooks.com/sampleBook.
                         spd"/>
               </cx:locator>
          </cx:digitalWork>
          <validityInterval>
               <notAfter>2006-12-24T23:59:59</notAfter>
          </validityInterval>
          </cx:print>
     </grant>
</license>
```

The ODRL [127] initiative is an international effort of supporters (Nokia, AegisDRM, etc) aimed at developing an open standard for a DRM expression language. ODRL is also a language based XML. The ODRL model consists of the following core entities:

— Assets include any digital content that can be uniquely identified.
— Rights include information consisting of the following:
  - Permissions
  - Constrains
  - Requirements
  - Conditions
— Parties include end users, roles and rights holders who can assert some form of ownership over the Asset and/or its Permissions.

With these three core entities, the foundation model can then express Offers and Agreements.

— Offers are proposals from rights holders for specific rights over their Assets (usually to end users).
— Agreements refer to contracts between Parties on the basis of specific Offers.

The following example illustrates that Mary Smith is purchasing a book and that she is given the rights to display and print it (there are two separate permissions specified in the example).

```
<?xml version="1.0" encoding="UTF-8"?>
<o-ex:rights xmlns:o-ex="http://odrl.net/1.1/ODRL-EX"
          xmlns:o-dd="http://odrl.net/1.1/ODRL-DD">
      <o-ex:agreement>
          <o-ex:context>
               <o-
          dd:uid>urn:ebook.world/999999/license/1234567890-
          ABCDEF</o-dd:uid>
```

```
     <o-dd:pLocation>Sydney, Australia</o-dd:pLocation>
     <o-dd:remark>Transacted      by      Example.Com</o-
dd:remark>
</o-ex:context>
<o-ex:asset>
     <o-ex:context>
          <o-dd:uid>urn:ebook.world/999999/ebook/rossi-
     000001</o-dd:uid>
     </o-ex:context>
</o-ex:asset>
<o-ex:permission>
     <o-dd:display>
          <o-ex:constraint>
               <o-dd:cpu>
                    <o-ex:context>
                         <o-dd:uid>Adobe-WebBuy:CPD-
                    ID:ER-393939-DSS-787878</o-dd:uid>
                    </o-ex:context>
               </o-dd:cpu>
          </o-ex:constraint>
     </o-dd:display>
     <o-dd:print>
          <o-ex:constraint>
               <o-dd:count>2</o-dd:count>
          </o-ex:constraint>
     </o-dd:print>
     <o-ex:requirement>
          <o-dd:prepay>
               <o-dd:payment>
                    <o-dd:amount
                     o-dd:currency=”AUD”>20.00</o-
               dd:amount>
                    <o-dd:taxpercent
                     o-dd:code=”GST”>10.00</o-
               dd:taxpercent>
               </o-dd:payment>
```

```
                    </o-dd:prepay>
               </o-ex:requirement>
          </o-ex:permission>
          <o-ex:party>
               <o-ex:context>
                    <o-dd:uid>urn:ebook.world/999999/users/msmth-
               000111</o-dd:uid>
                    <o-dd:name>Mary Smith</o-dd:name>
               </o-ex:context>
           </o-ex:party>
     </o-ex:agreement>
</o-ex:rights>
```

3. Security Requirements Verification

3.1. Overview

As we have discussed, system verification methods are broadly distinguished into static and dynamic verification methods.

Static verification methods aim to verify the satisfiability of specific properties by applying static analysis techniques to a program or a system specification without having to execute it. Static analysis techniques range from formal static verification to measuring the complexity or efficiency of algorithms.

In this section, we review techniques for performing static verification. More specifically we focus on formal methods that have analysed cryptographic protocols in order to detect security flaws. Also, we discuss some formal methods that have been used in the development of security systems.

3.2. Static Analysis of Cryptographic Protocols

A cryptographic protocol is a set of rules that describe how two or more agents can securely communicate over insecure open networks or distributed systems. Many different cryptographic protocols have been identified, and security systems usually adopt one or more of these. Static analysis is applied to cryptographic protocols in order to detect and remove security flaws and several different formal methods have been developed for this purpose.

In this Section, we first describe what a cryptographic protocol is. Then we identify two views of cryptography and briefly denote how they relate to one another. Moreover, we review the state-of-the-art of work in which cryptography is viewed symbolically. Finally, we describe some formal methods that have been used for developing security systems.

3.2.1. Cryptographic Protocols

A protocol is a series of steps, involving and taken by two or more active entities/roles, which is designed to accomplish a goal/task that is predefined by the involved entities/roles. Every step in a protocol must be executed in turn, and no step can be taken before the previous step is finished. Although each of the entities which are involved in protocol can alone perform a series of steps to accomplish a task the actions of one entity in isolation do not form a protocol. Furthermore,

— Every entity that is involved in a protocol must know it in advance.

— Every entity that is involved in a protocol must agree to follow it.

— The protocol must be unambiguous; each step must be well defined and there must be no chance of a misunderstanding.

— The protocol must be complete; there must be a specified action for every possible situation.

The execution of the protocol proceeds linearly through the steps, unless there are instructions to branch to another step. Each step involves at least one of two things: computations by one or more of the parties, or messages sent among the parties.

A cryptographic protocol is a protocol that uses cryptography. The parties can trust each other implicitly or they can be adversaries and not trust one another. One or more cryptographic algorithms can be involved during a cryptographic protocol, but generally the goal of the protocol is something beyond simple secrecy. The parties participating in the protocol might want to share parts of their secrets to compute a value, jointly generate a random sequence, convince one another of their identity, or simultaneously sign a contract. Cryptography is needed during the execution of a protocol in order to prevent or detect eavesdropping and cheating.

Depending on the roles of the entities which participate in it, a cryptographic protocol may be categorised in different categories. The first category is the arbitrated protocols. An arbitrator is a disinterested trusted third party whose participation is necessary in order to complete a protocol. The arbitrator has no vested interest in the protocol and no particular allegiance to any of the parties involved. The rest of the involved entities in the protocol accept the arbitrator's claims as true, its actions as correct, and that it will complete its part of the protocol. Arbitrators can help complete protocols between two mutually distrustful entities. An example of an arbitrated protocol is the time stamping protocol. In this case a party, P, wants to timestamp a digital document in order to certify that the specific document existed on a certain date. Using one-way hash functions and digital signatures we can provide a solution:

1. P produces a one-way hash of the document.
2. Then P transmits the hash to an arbitrator who is providing time stamping services.
3. The arbitrator appends the date and time it received the hash onto the hash and then digitally signs the result.
4. The arbitrator sends the signed hash with the timestamp back.

Because of the high cost of hiring arbitrators, arbitrated protocols can be subdivided into two lower-level sub-protocols. One is a non-arbitrated sub-protocol, executed every time parties want to complete the protocol. The other is an arbitrated sub-protocol, executed only in exceptional circumstances—when there is a dispute. This special type of arbitrator protocol is called an adjudicator protocol. An adjudicator is also a disinterested and trusted third party. Unlike an arbitrator, it is not directly involved in every protocol. The adjudicator is called in only to determine whether a protocol was performed fairly. For example, Alice and Bob might draw up a contract agreeable to both of them and sign it. Both keep a copy but later, if there is a dispute, both can present their evidence before an adjudicator.

Finally the last type of cryptographic protocol is the self-enforcing protocol. A self-enforcing protocol is the best type of protocol because the protocol itself guarantees fairness. Neither an adjudicator nor an arbitrator is required to resolve disputes or to complete the protocol. The protocol is constructed so that there cannot be any disputes. If one of the parties tries to cheat, the other party immediately detects the cheating and the protocol stops. Whatever the cheating party hoped would happen by cheating, doesn't happen. This is the most desirable type of protocol, but also more difficult to achieve, and such protocols are not known for all problems.

3.2.2. *Symbolic vs Computational View of Cryptography*

There are two different views of cryptography [2]: the symbolic and the computational view.

The symbolic (or formal) view of cryptography treats cryptographic operations as "purely formal", i.e. described using a formal notation or logic. For example, for an encryption protocol the encrypted message, the key and the plaintext are all expressed as formal expressions. Operations are then considered as computations on expressions, usually generating other expressions. The security properties of encryption are built into the model that describes the system's behaviour. The symbolic view of cryptography includes a variety of techniques and approaches from the fields of rewriting, modal logic, process algebra, formal methods, and others. Initial work in this area includes work by Dolev and Yao [87], Kemmerer [136], Burrows et al. [48] and Meadows [174].

The computational view of cryptography is based on the framework of computational complexity theory. In an encryption protocol, for example, the encrypted message, the key and the plaintext are all strings of bits. And operations as, for example, the encryption function are simply algorithms. Abadi and Rogaway [2] indicate that the adversary is treated a Turing machine. The aim of approaches and techniques that adopt the computational view of cryptography is to measure how good a protocol is in terms of probabilities and computational cost. A protocol is good if the adversary does not do "something bad" too often or too efficiently. Work in this area was initiated by Yao [254], Blum and Micali [38] and Goldwasser and Micali [103].

Abadi and Roagaway [2] show that there are connections between the symbolic and the computational view whose investigation should benefit both. These connections can, for example, help clarify any implicit assumptions or gaps in the formal methods. Methods for high level reasoning that are used to analyse complex systems seem necessary for computational cryptology.

In this deliverable, we focus on the symbolic view of cryptography and in particular with formal methods that help us to detect security flaws.

3.2.3. Formal Methods

Formal methods describe a group of mathematically based approaches for specifying, designing, analysing and verifying systems. In the 70s and early 80s, the National Security Agency in the United States was a major source of funding for formal methods research and development, which resulted in the development of formal models of security systems and tools for analysing system security properties and proving that system are secure.

In the following sections, we review the various methods for formally analysing cryptographic protocols as classified by Meadows [289]:

— **General purpose verification methods and tools:** These consist of specification languages and verification tools that were not specifically developed for statically analysing cryptographic protocols, but they have been used to do so. These include model checkers and theorem provers, and methods that combine both.

— **Expert systems:** A protocol designer may use expert systems to explore different scenarios.

— **Modal logic based approaches:** These approaches use modal logics based on knowledge and belief for modelling and verifying protocols.

— **Algebraic based approaches:** These develop a formal model based on the algebraic term-rewriting properties of cryptographic systems.

Moreover, we present formal methods that have been used for developing and analysing security systems, which can include cryptographic protocols, but not necessarily, and their security properties.

3.2.3.1 General purpose verification methods and tools

General purpose static verification methods treat cryptographic protocols as distributed systems and try to prove their correctness. Firstly, the protocol and its desired properties (in this case security properties) are specified in the specification language of the method, such as finite state machines, Communication Sequential Process (CSP), Petri Nets. Subsequently, tools are used for verifying the properties, such as model checkers, theorem provers or even a combination of both.

Model checkers analyse cryptographic protocols by exhaustively exploring the state space of a model in order to verify its desired properties. The properties are usually specified using temporal logic (usually Linear Temporal Logic or Computation Tree Logic or a variation of these). If a property is found to be false, a counter-example is produced outlining the steps that lead to the contradiction. Model checkers, however, suffer from the state space explosion problem, which can make model checking of large systems infeasible. To deal with this problem, researchers have developed symbolic algorithms, partial order reduction techniques, on the fly model checking and abstraction techniques [62].

The CSP/FDR framework for the analysis of security protocols was initiated mainly by Gardiner, Goldsmith, Jackson, Lowe and Roscoe [206, 207]. The process algebra CSP is ideal for modelling security systems because it can describe systems that are composed of parallel processors communicating with one another by synchronisation of some common events. Roscoe and Goldsmith [290] describe how a cryptographic protocol attacker can be modelled by the CSP inference system. Once modelled in CSP, a security protocol can be verified using the FDR2 tool. FDR2 uses a lazy exploration strategy to examine subset of intruder states which are reachable by the protocol rules. Thus, FDR2 examines the behaviour of the intruder together with that of the protocol's. An advantage of this approach is that it is able to reason about the absence of denial-of-service attacks (liveness properties). Usually the CSP code is derived by hand which can be time-consuming and error-prone. A program called Casper [163] was developed to convert a high level description of a protocol (written in a simple language for describing protocols) into CSP code.

Murφ [291] is a general-purpose state enumeration tool that was used by Mitchel et al. [292] to analyse security protocols. The approach of Murφ is similar to that used in CSP model checking. The protocols are described using the Murφ language which is a simple high-level language for describing non-deterministic finite-state machines. The properties are specified by invariants, i.e. Boolean conditions that have to be true in every reachable state (hence, no temporal operators). Murφ uses breath-first or depth-first full state enumeration for verifying that all reachable states of the system satisfy the desired properties. The main method adopted for analysing protocols consists of formulating the protocol in the Murφ language, adding an adversary to the system (the adversary is allowed to overhear, intercept and generate messages), stating the desired correctness condition (this has not proved to be difficult in the protocols they described, but for others protocols it could be), running the protocol with some specific size parameters, experimenting with alternative formulations, and repeating the steps. Known flaws of the Needham-Schroeder Public-Key [192], TMN [293] and Kerberos [294] protocols have been identified using this approach. The difficulties encountered with Murφ include modelling the adversary and formalising its "knowledge", and selecting a finite set of possible adversary actions at any point in the run of the protocol using the adversary's knowledge at that point. An advantage of Murφ is that it is possible to change a system description to reflect a situation where one or more items of secret information have been compromised.

ASTRAL [295] is a formal specification language for specifying real-time systems. ASTRAL consists of global and process specifications. Global specifications consist of declarations of process instances, global constants and non-primitive types that may be shared by process types, and system level critical requirements (properties that will be verified). ASTRAL specifications also include environmental assumptions that formalise the assumptions that must always hold on the behaviour of the environment to guarantee some desired system properties. The system properties are verified by the ASTRAL model checker. Since ASTRAL is undecidable, certain restrictions occur in order to be able to model check specifications in it. For instance, ASTRAL specifications are based on infinite state machines with unlimited time bounds and for model checking to be feasible, a finite time bound needs to be set by the user. ASTRAL was applied to the Needham-Schroeder public-key authentication protocol and the TMN protocol [296]. One of the flaws of the TMN protocol was missed by the ASTRAL model checker because it required excess time bound for the specific ASTRAL modelling of the protocol. This same flaw was detected by both FDR and Murφ. However, these results were preliminary. More recently, Dang and Kemmerer [297] analysed a more complex time-dependent protocol, called Mobile IP, that can be considered as a real-time protocol unlike Needham-Schroeder and TMN. For example, a timing property would require that a mobile node needs a time reference in order to decide whether its current registration is going to expire, and a timestamp mechanism to protect against potential replay attacks. The ASTRAL model checker detected specification errors and once these were corrected, no flaws in the protocol were discovered.

As mentioned above, model checking suffers from the state space explosion problem. In order to address this problem, Bolignano [39] introduced an approach for generating human-readable proofs that can be used as part of a vulnerability analysis or formal code inspection. Specific properties of the problem are used to formalise the requirements and simplify the proofs. The conciseness of the verification process is comparable to that of the modal logic and this is because of the use of powerful invariants and the axiomatisation of the intruder knowledge. The Coq proof assistant [31] can be used to automate this process within a framework of typed logics.

Schneider [216] describes an approach for the analysis and verification of authentication properties in CSP. The CSP syntax can describe authentication protocols precisely in terms of the messages accepted and transmitted by each participant in the protocol. Schneider’s aim was to build a separate theory for analysing authentication protocols on top of the general CSP framework. This theory has been successfully used to model and verify the Zhou and Gollman fair non-repudiation protocol [257].

Kemmerer [136, 137] introduced an approach based on an extension of first order predicate calculus, that uses the Ina Jo specification language. Ina Jo is a tool that was designed to support the development of software that includes correctness proofs. The security system is expressed as an Ina Jo specification and so are the security properties that must hold in all states. Verification occurs with the generation of theorems that are used to prove the properties. This approach detected a security flaw in the system. However, Buttyan [49] points out that since the designer needs to know the potential attacks in advance, the benefits of this approach are limited.

Nieh and Tavares [195] use coloured Petri nets for modelling and analysing cryptographic protocols. A general intruder model is also included in the overall model in order to represent intruder attacks and generate test cases. The security properties are analysed by performing an exhaustive penetration test that searches for scenarios that violate certain specified criteria. These criteria are defined in terms of requirements on Petri net states. This approach suffers from the lack of available tools for automating the exhaustive search. A solution to this problem would be to translate the coloured Petri nets to ordinary Petri nets and use the available tools. However, like model checkers, these tools suffer from the state explosion problem.

More recently, Dai et al. [298] use the Software Architecture Model (SAM), a general formal framework that is based on Petri nets and temporal logic, to specify and analyse authentication protocols. In particular, they use Predicate Transition nets [100] and first order linear temporal logic [167] to model the registry protocol [109]. Each involved participant, including trusted principles and intruders, is explicitly modelled along with its behaviour and interactions. The SPIN model checker [124] is used for verification. Dai et al [298] describe a process that can be automated, i.e. they provide a systematic method for translating a SAM model into a SPIN specification. The main advantage of using this approach is that the graphical representation (Petri nets) enhances understanding and its well-defined semantics facilitates analysis. This approach identified similar security flaws as those identified by the LOTOS [155] when applied to the same case study.

Other recent work in the area of Petri nets includes: Al-Azzoni et al. [7] who propose a technique for modelling and verifying cryptographic protocols using coloured Petri nets and Design/CPN and illustrate it on the TMN protocol; Aly and Mustafa [11] who analyse and verify the STS protocol [84] using coloured Petri nets; and Crazzolara and Winskel [74] who show how Petri nets can be used to prove security systems (these researchers present a process language together with semantics for security protocols).

3.2.3.2 Expert Systems

The Interrogator [179, 180] is a software tool written in Prolog that explores the state space of a model exhaustively in order to identify any security flaws. Interrogator is one of the first systems that uses the Dolev-Yao approach. The abstract model includes the usual state variable for the intruder's set of known items. However, this known set is not explicitly mentioned in the state representation used by the recursive search algorithms. The Interrogator models the protocol's participants as communicating state machines whose messages are intercepted by an intruder. The intruder can destroy messages, alter them, or let them pass through unaltered. Assuming a final state where the intruder knows some word that should be kept secret, the Interrogator explores all the possible paths through which it can reach that final state. If a path is found, then a security flaw is identified. However, if no path is found there is no guarantee that no attacks can exist in the system model. The Interrogator has identified only previously known attacks on the protocols analysed.

The NRL Protocol Analyzer [137, 174, 175] is comparable to the Interrogator as it attempts to construct a path from an insecure state, given by the designer, to the initial state. The main difference between the NRL and the Interrogator is that NRL not only tries to find paths to the insecure states, but also proves that these states are unreachable. The proofs show that certain paths that lead backwards from the insecure states end up in infinite loops, and hence never reach the initial state. These paths can be eliminated, thus reducing the search space that is explored exhaustively. However, the proofs require user guidance making the search less automated than that of the Interrogator. Another difference with the Interrogator is that the NRL Analyzer can construct a single path using an arbitrary number of protocol rounds, thus working in an infinite state space. Therefore, the NRL Analyzer can detect attacks based on a combination of protocol runs. The NRL Protocol Analyzer was successful in identifying a number of unknown flaws in several protocols [220, 47] as well as identifying known flaws.

Longley and Rigby [162] have developed a rule based system that decomposes goals into subgoals, and then subgoals are decomposed further and so on to build a tree. The root node of the tree represents the data item required by the intruder for an attack and the leaf nodes represent the data that must be known in order to know the root item. The user can interact with the system to determine whether or not a data item can be found by the intruder, even if the system has classified that data item as being inaccessible to the intruder. If a data item is found to be accessible to the intruder, it is added to the system and a tree is generated for it. The search tool developer [162] identified a subtle flaw of the hierarchical key management scheme. Its approach is similar to that of the Interrogator; however it relies on user interaction.

Expert systems developed specifically for the analysis of cryptographic protocols can be more successful than general purpose tools in detecting unknown flaws. However, they suffer from the state space explosion problem, which can lead to systems never halting and inconclusive results. To handle this problem, user interaction is required, which means that the search is not completely automated. The strength of expert systems lies in the fact that if they detect a flaw, then the attack scenario is directly available, which is not the case for modal logic based approaches.

3.2.3.3 Modal logic based approaches

A modal logic approach consists of a language that is used to describe the cryptographic protocol as logic statements expressing what the participants know and believe, and some inference rules for deriving new statements. The purpose of analysis is then to derive a statement that represents the correctness condition of the protocol. This derivation may often reveal flaws in the protocol.

BAN logic [48] is the most widely used formal logic for analysing authentication protocols. It belongs to the class of KD45 modal logics, which means that any fact is only a belief and does not need to be universal in space and time. Ban logic analyses protocols by firstly, expressing the assumptions and goals as statements in a symbolic notation in order for the logic to proceed from a known state to one in which if can check whether the goals can be reached. Subsequently, the protocol steps are also formalised into the symbolic notation. Finally, postulates, which are a set of deduction rules, are applied and these should lead from the assumptions to the authentication goals, via intermediate formulas.

The use of Ban logic has revealed flaws in several protocols, including Needham-Schroeder [192] and CIT X.509 [55]. It has also uncovered redundancies in Needham-Schroeder, Kerberos [181], Otway-Rees [196], and CCIT X.509. In spite of its success, however, BAN logic has been criticised by various researchers. This criticism relates, for example, to the difficulty of BAN logic in proving completeness properties [159], its limitation of providing only partial correctness proofs [223] and its inability to discover flaws which violate basic security requirements of authentication [194]. The most crucial weaknesses of BAN logic, which have been discussed in the literature, are that there is no complete semantics for the logic and when modelling freshness, it is not possible to distinguish between freshness of creation and freshness of receipt. The lack of complete semantics can lead to problems when formulating the BAN specification of an informal protocol description (this process is called "idealisation") due to vagueness and ambiguity.

Abadi and Tuttle [3] overcome the problem with idealisation by reformulating the original BAN logic and providing new semantics for it. The changes made include removing unnecessary mixing of semantic and implementation details in the definitions and inference rules, defining concepts such as seeing, believing, etc. independently rather than jointly with other concepts, and reformulating the set of inference rules as an axiomatisation with modus ponens and necessitation being the only rules. The result is a much simpler logic which was claimed to be sound with respect to the new semantics although Syverson and van Oorschot [230] later identified an axiom of the logic that was not sound.

Many extensions of BAN logic have been developed as researchers aim to improve on its limitations. One of these successful extensions is the GNY logic [104] which extends the scope of BAN logic but is more complicated. GNY logic analyses a protocol step-by-step, expresses any assumption required explicitly and draws conclusions about the final position that it has arrived at. GNY logic improves on the BAN logic in the following ways:

— It separates the content and the meaning of messages, therefore increasing consistency in the analysis and it is possible to reason in more than one level.

— Message data can include principles even if they don't believe in them.

— The ability of a recipient to identify the expected message can be expressed and one is allowed to determine that some messages are not replays of the recipient's earlier messages given in a session.

The main drawbacks of GNY logic are that it addresses only authentication and many of its rules have to be considered at each stage, thus making it more complicated than other methods [13].

BGNY [41] is an extended version of GNY logic that has been formalised with a Higher Order Logic (HOL) [105] theory. As with GNY logic, BGNY focuses only on authentication. The authentication properties of cryptographic properties are proved automatically with HOL software. BGNY differs from GNY as it is able to specify properties at intermediate stages and it is able to specify protocols that use multiple encryptions and hash operations, message authentication codes, and has codes as keys and key-exchange algorithms.

Other extensions of BAN logics include:

— An extension by Mao and Boyd [169] whose work does not cover protocols using public-key algorithms nor does it include theoretic proof soundness of the proposed idealisation rules;

— An extension by Gaarder and Snekkenes [99] which can reason about time;

— An extension of BAN and GNY [241] which handles key agreement protocols such as Deffie-Hellman; and

— An extension by Campbell et al. [52] which supports probabilistic reasoning for calculating a measure of trust rather than complete trust.

The above list is not a complete account of work in this area and several other extensions have also been proposed. These extensions, however, are beyond the scope of this survey which focuses on dynamic verification.

SvO is another logic [230] that encompasses the features and reasoning of four logics, namely BAN, BNY, Abadi-Tuttle logic and vO [241], in a unified framework. Syverson and van Oorschot define model-theoretic semantics for SvO with respect to which the logic is sound. SvO is considered to be easier to use and more expressive than the four logics it was derived from.

Other logics for static verification which are not extensions of BAN logic include:

— Rangan's logic [203], that can be used to reason about the effect of trust in the composition of secure communication channels and provides a formal basis for the evolution of belief from trust.

- Moser's logic [185] that is a non-monotonic logic used to reason about beliefs of protocol participants and how these beliefs change (e.g. in cases when a key used in a secure communication is compromised).
- Bierber's CKT5 [37] that is used to reason about the evolution of knowledge about words used in a cryptographic protocol and makes a distinction about seeing a message and understanding its importance.
- Syverson's KPL [229] that is used in the same way as CKT5.
- The Yahalom et al. system [252] that is used to derive information about the nature of the trust that protocol participants must have in each other in order for a protocol to operate correctly.
- Kailar's logic [132] that is used for the analysis of communication protocols that require accountability, for example, for secure electronic transactions. This logic is based on the AUTLOG semantics [139].
- Wedel and Kessler logic [247] that is used for the analysis of authentication protocols and provides formal semantics for proving its soundness. A wide variety of cryptographic mechanisms can be described in this logic using the most concise notation.

3.2.3.4 Algebraic approaches

Algebraic approaches model a protocol as an algebraic system and use the algebra to formalise the state of each participant's (including the intruder) knowledge about the protocol. Research in this area is not as active as research in developing the other formal approaches. Nevertheless, algebraic models have shown to be successful when representing subtle kinds of knowledge in cryptographic protocols.

Dolev and Yao [87] were the first to model a cryptographic protocol as an algebraic system. In their model, the intruder has control of the network and can read all the traffic, modify and destroy messages, and perform any operation, such as encryption, as an authorised user would. Initially the intruder does not know any of the secret information, such as the authorised user's keys. Furthermore, Dolev and Yao treat every message sent by an authorised user as if it was sent to the intruder and every message received by the authorised user as if the intruder sent it. This is because of the intruder's control over the network. Therefore, the system becomes a machine used by the intruder to generate words which obey certain re-write rules, for example a rule would be that encryption and decryption with the same key cancel each other out. If the intruder's aim is to discover a word that is secret, the problem of proving a protocol secure is the same as the problem of proving that a word cannot be generated in the term rewriting system. This observation is used to develop algorithms to analyse the security of certain classes of public key protocols, namely cascade protocols and name-stamp protocols.

The Dolev and Yao approach is limited and cannot be useful for analysing a wide range of protocols for the following reasons. Firstly, it can only detect failures with respect to secrecy, and participants do not remember state information from one state to the next. Some research aims to overcome these problems and find ways to analyse other classes of protocols. These include, Merrit [177] who generalises the Dolev and Yao approach to model diverse cryptographic systems and formally proves other properties besides secrecy, and Toussaint [236] who describes a technique, which is based on Merrit's [177] algebraic model, and can derive the complete knowledge of each participant in the protocol.

Recently, Abadi and Gordon [1] use the *pi* calculus to describe protocols at an abstract level. Properties of cryptographic protocols are modelled using pi calculus primitives for channels, in particular scoping rules. Pi calculus is further extended to what is known as *spi* calculus for analysing the protocols at a lower level of abstraction. The security properties of the protocol are expressed in *spi* calculus as equivalences between *spi* calculus processes. For example, the protocol keeps secret a piece of data X if the protocol containing X is equivalent to the protocol containing X' for every X'. The intruder is not explicitly modelled, but is represented as an arbitrary *spi* process. This is an advantage as modelling the intruder can be difficult and error prone.

3.2.4. Developing security systems with formal methods

The Software Cost Reduction (SCR) method [299] is a set of techniques used for developing systematically formal specifications from a set of requirements. SCR was developed from a collaboration between David Parnas and Constance Heitmeyer, and other researchers from the U.S. Naval Research Laboratory (NRL) in the late 1970's and consists of a tabular notation that is based on state machines. SCR specification properties are written as logical formulae and divided into static (no temporal operators) and transition properties (with temporal operators). SCR is supported by tools that have been developed for creating and validating SCR specifications, and analysing properties such as syntax and type correctness, case coverage, determinism and lack of circularity. The SCR toolset includes the model checker SPIN [124], the verifier TAME (i.e. a user-friendly interface for the PVS theorem prover [300]), a property checker based on decision procedures called Salsa [301], and an invariant generator. These tools can assist the verification of critical safety and security properties.

SCR has been applied to a Communications Device (CD) that provides cryptographic processing for a U.S. Navy radio receiver and to a biometrics standard (BioAPI) [302]. As it has been reported, it took about one-person month to produce the SCR specification of CD (which was moderately complex) and to verify seven security properties. Developing a formal specification in SCR from the requirements document (expressed in prose) was very difficult as it was structured differently. This brought about some questions concerning the requirements and the use of tools such as the simulator and invariant generator, further errors and missing cases in the specification were identified. Tame was used to verify the security properties, which required supporting invariant lemmas that were obtained from the set of invariants produced by the invariant generator. There were problems verifying an eighth property in Spin because of the state space explosion problem.

It took about two weeks to specify the BioAPI standard in SCR and to check that there were no missing cases and ambiguities, and to verify one critical authentication property. In fact, as reported in [302], the correct formulation of this particular security property was very difficult and it took a lot more time that it took to verify it.

An open problem, discussed in [302], was how to validate the source code that implements a secure system. Even though the security properties are verified at the specification level, it is still necessary to demonstrate that the source code operates securely. Heitmeyer [302] suggests an approach where one can derive a set of test cases from the specification and use these in order to test that the source code satisfies the specification. Some initial work on this is presented in [303].

The B-method is a formal method developed by Abrial [5] and is supported by two tool suites that have been used to develop industrial applications: the B-toolkit [35] and the Atelier B toolkit [20]. B specifications are expressed using the *Abstract Machine Notation* (AMN) that is based on first order predicate logic and set theory. Temporal properties cannot be proven easily in B. To overcome this problem, Abrial and Mussat [304] introduced new clauses in the B notation to express temporal properties of event-driven systems. However, the manner in which these properties are expressed is complex and does not resemble their logical form. The overall development process that is based in this approach takes place through stepwise refinement. In this process, proofs have to be produced in each refinement to ensure consistency.

A specific refinement technique was developed by Abrial for verifying security protocols. The basic idea is to describe the objective of the protocol in a single instantaneous operation along with a number of void (skip) operations that will be refined one by one in the following refinement steps, until the entire protocol is specified. By verifying each refinement and because of the transitivity of refinement, the entire development of a protocol is eventually verified. Compared to CSP, the B-method is more state-based and therefore, it makes coding easier. However, it is not well-suited to dealing with concurrency.

Recently, the B method has been used to model protocols for Java smart cards. The problems in this area are that Java cards have small memory and weak CPU, which consequently means that any security run-time checks must be minimised. Girard and Lanet [308] use the B method to model several parts of the Java card as well as the virtual machine. The main objectives for using B method for the design of the virtual machine are to:

— Express the virtual machine formally;

— Extract the static checks;

— Show formally that the interpreter satisfies the static constraints; and

— Provide an implementation of both the verifier and the interpreter.

By applying this approach Girard and Lanet [308] were able to reduce the amount of run-time checks required. Also, they proved mathematically that the implementations of the interpreter, firewalls and backup mechanism (Operating System functionality) are correct, hence guaranteeing a secure platform. Other work with respect to developing smart cards includes [305, 306, 307, 308, 310].

Hall and Chapman [311] describe how they developed from requirements to code a commercial secure system, in particular a Certification Authority (CA) system, that had to meet stringent security requirements as well as normal commercial requirements (throughput, usability, cost). Hall and Chapman used a requirements-engineering method called Reveal [113], to define the CA's environment and business objectives, and map them into system requirements. The user requirements were written in English with context diagrams, class diagrams and structured operational definitions. Also, each requirement was labelled in order to be able to trace it back to its source. Security requirements would be traced back to the corresponding threats. Although the ITSEC (www.cesg.gov.uk) require Formal Security Policy Models, the requirements included an informal security policy that identified threats, assets and countermeasures. Hall and Chapman reported that they managed to formalise only 23 items of the 28 technical items.

In their case study, Hall and Chapman also used Z [225] to specify the modules that manage the cryptographic keys and their verification on system start-up. What they found was that Z was not a good language for expressing information separation, and that CSP was a better language for this. They did not carry out any proofs of correctness at this stage, only used a typechecker. They modelled the process structure in CSP, by mapping sets of Z operations to CSP actions. Checks were carried out to ensure the system was deadlock free and that there was no concurrent processing of security functions. Translation rules were devised for mapping the CSP model to code (Ada95 and Spark which is a subset of Ada95). Static analysis was carried out on the Spark code, which has additional annotations for performing data- and information flow analysis, and to prove properties of code, such as partial correctness and freedom from exceptions. A Spark program is checked to be free of any dataflow errors (such as the use of an initialised variable) that can lead to subtle security flaws. Thus, proof of correctness of Spark programs with respect to its formal specification can be achieved.

3.2.5. Specification of security and other system properties for static verification

Table 3.1 gives a summary of various formal notations which have been used by different static verification methods to express the properties to be verified and other functional and non functional characteristics of the systems and identifies languages and notations that have been specifically developed for expressing and verifying security properties.

The security properties are usually expressed in some formal notation, chosen according to its expressivity and which verification approach is going to be used. For static verification of cryptographic protocols, BAN logic is most commonly used for formalising security properties. System descriptions (specifications) are also expressed using a formal notation, such as finite statemachines, Buchi automata or even CSP. Specifications can be defined at different levels of abstraction. For example, KAOS defines specification at a very high level of abstraction, i.e. in the requirements capturing phase. The B-method [5] allows one to express the specification initially at a high level of abstraction and then with a series of refinement steps, refines the specification to a low level of abstraction, where code can be generated automatically.

Languages for static verification of cryptographic protocols	Languages for static verification of security systems
BAN logics [48]; or various extensions of BAN logics	SCR tabular notation & properties expressed as logic formulae [299]
HOL [105]	B-method [5]: Abstract Machine Notation – based on first order logic and set theory
SvO [230]	Z [225]
Coq [31]	
CSP/FDR [206]	
Petri Nets [195]	
Astral Model Checker [295]	
Murφ langauge (based on finite-state machines) & properties are Boolean conditions that have to be true in every reachable state [291].	

Table 3.1 – Summary of formal languages used for static verification

3.2.6. Conclusions for static verification and formal methods

Although the use of formal methods and static verification have made significant contribution in the verification of security properties and the development of security solutions, the survey of the literature in this area has identified that these approaches have certain limitations. These limitations can be summarised as follows:

- As pointed out in [250], systems do not run in isolation but operate in typically complex environments. Thus, the formal specification of a system must always contain the environmental assumptions and a proof of correctness is valid only if these assumptions hold. It is, however, impractical to state every environmental assumption explicitly and some will inevitably be missed. Also, attackers can take advantage of the explicit modelling of such assumptions and find out how to violate them. Furthermore, even if one had thoroughly expressed each assumption, it is still possible that a system may eventually be deployed in a different environment in which the original environmental assumptions may not hold.
- Certain security properties are difficult to model and this reduces the applicability of use formal methods in large scale validation.
- The formal specification of security properties of complex systems, other behavioural properties of these systems which interact with the security properties and the interaction of

the systems with other systems in their environment may grow to a size that makes formal verification intractable due to the state explosion problem. This problem has been identified in almost all the reviewed static verification techniques which tend to focus on the verification of single security properties or small sets of such properties.

- Security is a combination of properties, which may be satisfied at varying degrees in different environments, rather than being completely satisfied or dissatisfied. Also, certain properties are more important than others and some might even be in conflict with each other. For example, in electronic payment systems, client anonymity and accountability of could both be desired albeit conflicting properties. Formal methods and static verification are not very well-suited to support analysis for graded satisfiability of properties of varying importance and possibly conflicting.
- Even if static verification can prove that the specification of a system satisfies certain security properties, there is no guarantee that the implementation of the system will be compliant with the specification and, therefore, the proved properties will also hold during the operation of the system.

Finally, Meadows [176] has pointed out that certain open ended issues emerge from the formal analysis of cryptographic protocols. These issues include: how to model open-ended protocols with no fixed number of participants and unbounded length of messages, identifying new threats and modelling new types of applications, identifying security flaws at lower levels of abstraction and how can systems that use a composition of cryptographic protocols be verified.

4. Dynamic Verification

Dynamic verification enables a software system to improve its dependability (and therefore security) [21], by checking whether its behaviour satisfies specific dependability and security properties while it is running. This can be accomplished by a software module, which monitors the execution of the system and checks its conformity with the specification of the relevant properties. This module can be either an external or an internal module of the monitored system.

Software systems are increasingly becoming ubiquitous and heterogeneous and rely on technologies such as mobile code and components off the shelf (COTS). Static verification and testing of dynamically adapted entities cannot provide adequate results, each one for different reasons. Static verification is a formal method and can prove that a system (or to be more accurate its model) is correct but is very time consuming and demands substantial education and experience from practitioners. Testing [156] on the other hand is an informal method which cannot prove a system correct since it can never offer a complete coverage of all its possible executions but can be easily applied even from inexperienced practitioners.

Being situated somewhere between static verification and testing, dynamic verification techniques aim to achieve the benefits of both approaches, by merging testing and formal specification. Thus, dynamic verification is considered to be a formal method applied to the implementation of the system that avoids the pitfalls of ad hoc testing and the complexity of full blown static verification techniques (model checking, theorem proving).

According to the literature on dynamic verification [32, 78, 119], the basic stages of dynamic verification are: (i) the development of a formal specification of a system including various types of properties, like safety and security properties, (ii) the application of methods for capturing events of interest and (iii) checking for violations by a monitor which can verify whether the observed behaviour of a system satisfies the required properties.

It should be noted that there are cases such as Aspect Oriented Programming [140] and Monitoring Oriented Programming [59] in which a monitor is generated automatically and inserted into the code that has to be monitored. Thus, in such cases, the second stage includes the monitor generation as well. On the other hand, in all the other cases, monitors are considered to be software modules, which have to be implemented [19, 119] separately from the monitored system. The monitor inputs are the formal specification of the system (product of first stage) and the flow of events generated during the execution of the system. The monitor then reasons about the conformance of the captured runtime behaviour of the system (events flow) against the indented system behaviour (formal specification).

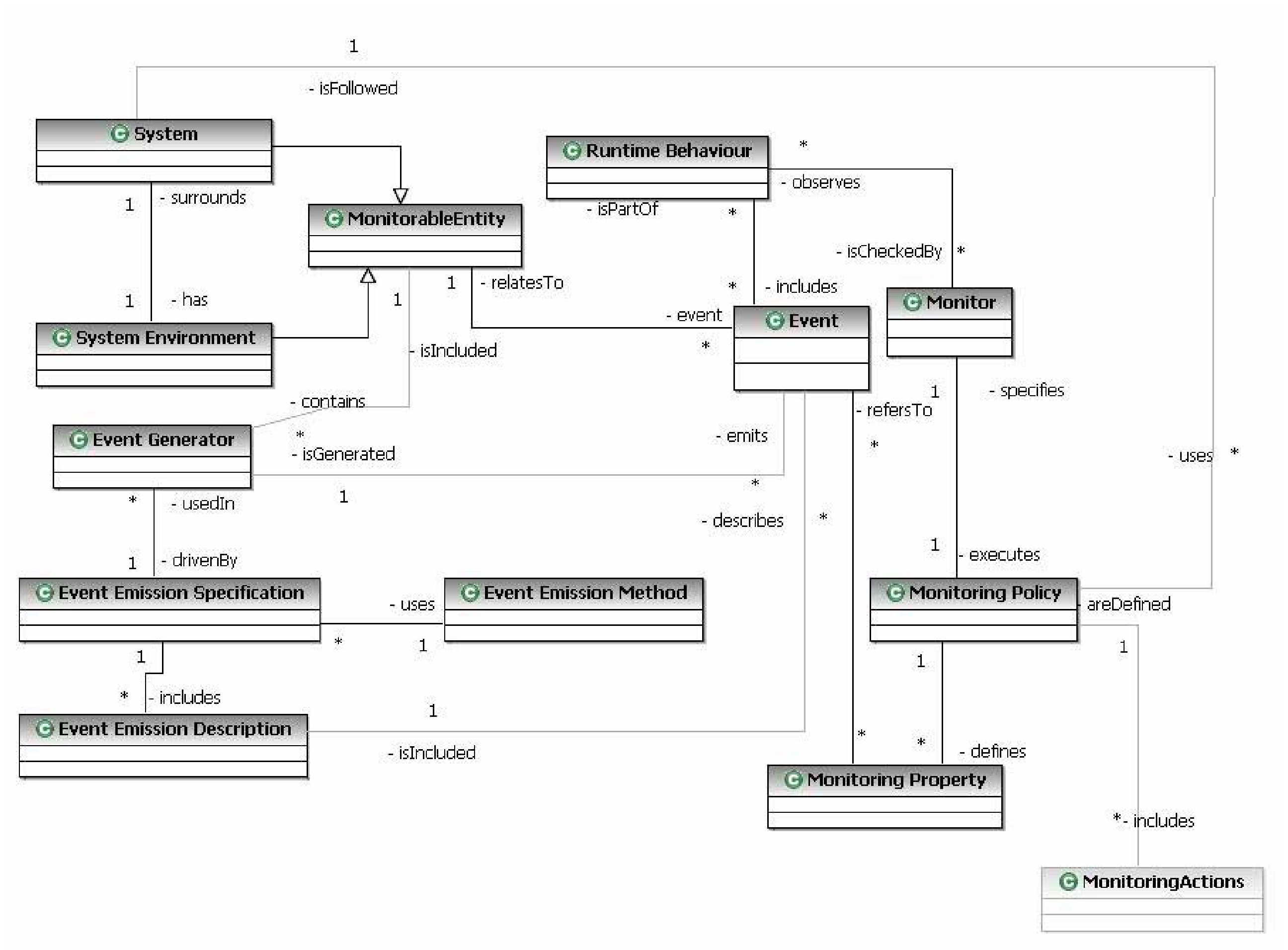


Figure 4.1 – Conceptual Model for Dynamic Verification

Figure 4.1 shows the conceptual model we have constructed to indicate the entities involved in dynamic verification. According to this model, the subject of dynamic verification that is signified by the class *MonitorableEntity* can be either a *System* or a *System's Environment*. Dynamic verification is carried out by a *Monitor* which observes the *Runtime Behaviour* of a system or its environment. The *RuntimeBehaviour* is a set of *events* generated during the operation of the monitorable entities. These events are generated by one or more *Event Generator* according to different *Event Emission Specifications*. An event emission specification describes the particular *Event Emission Method* to be used and one or more *Event Emission Descriptions*, which describe the exact types of events which should be generated. The observation of the events in a *Runtime Behaviour* by the *Monitor* is carried out according to a specific *Monitoring Policy* which specifies the *Monitoring Properties* that should be verified at runtime and the set of *Monitoring Actions* the *Monitor* should perform to enable the system control and/or recover from violations of the monitoring properties.

Figure 4.2 presents taxonomy of monitor and event generation features. This taxonomy has three layers which differentiate monitoring and event generation capabilities according to (a) the controlling capabilities of a monitor, (b) the time of the event emission with respect to the occurrence of the action described by the event, and (c) the communication type between the monitor and the system.

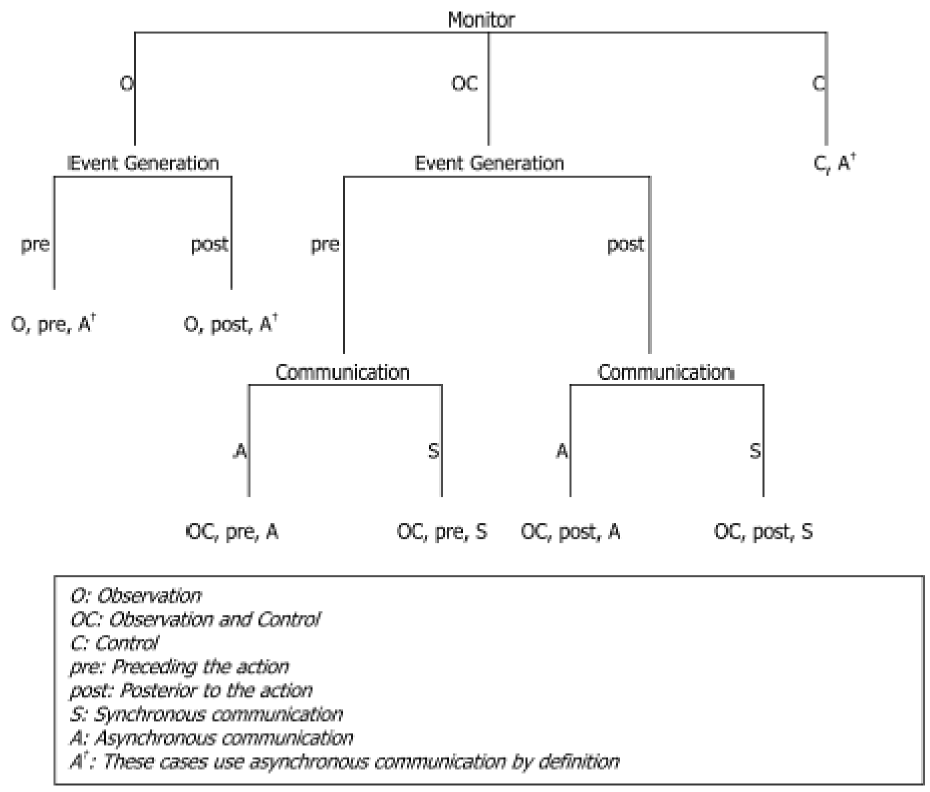

Monitor
O
OC
C
Event Generation
Event Generation
C, A†
pre
post
pre
post
O, pre, A†
O, post, A†
Communication
Communication
A
S
A
S
OC, pre, A
OC, pre, S
OC, post, A
OC, post, S
O: Observation
OC: Observation and Control
C: Control
pre: Preceding the action
post: Posterior to the action
S: Synchronous communication
A: Asynchronous communication
A†: These cases use asynchronous communication by definition

Figure 4.2 – Taxonomy of Monitor and Event Generation Features

More specifically at the first layer a distinction is made based on whether the monitor has observation only, observation and control or control only capabilities. These capabilities can be summarised as follows:

- Observation (O): The monitor observes the runtime behaviour of the system by receiving the generated events and it checks whether the monitoring properties hold at runtime.
- Observation and Control (OC): The monitor observes the runtime behaviour of the system by receiving the generated events, it checks whether the monitoring properties hold at runtime and forces the system to execute specific actions. These actions can be either preventive or perform recovery. This class is also known as closed-loop control.
- Control (C): The monitor forces the system to execute actions without needing to observe the actual state of the system. This class is also known as open-loop control.

The second layer of the taxonomy presents a distinction according to the time of the event emission with respect to the occurrence of the action described by the event. According to the criterion, we can distinguish between two cases:

- Emission preceding the action (pre): The event precedes the action which it describes. For example, the event generator sends an event to the monitor informing it that the system wishes to lock some resource before the system locks it.
- Emission posterior to the action (post): The event follows the action which it describes. For example, the event generator sends an event to the monitor informing it that the system has completed some transaction.

Finally, the third layer of the taxonomy refers to the type of the communication between the monitored system and the monitor. According to this criterion, we distinguish between the following two types of communication:

- Synchronous communication (S): The event generator uses a blocking send primitive to communicate with the monitor, waiting for a reply from it. This is only used when the monitor can exert control over the system.
- Asynchronous communication (A): The event generator uses a non-blocking send primitive to communicate with the monitor. It is mainly used when the monitor cannot exert any control over the system or when the control actions can be applied asynchronously. For example, the monitored system may notify the monitor that it will attempt to perform some action and start performing it without waiting for a permission to do so, as in optimistic transactions. If the monitor subsequently decides that this action is undesirable it can send a signal to the system to abort the action.

4.1.1. Formalisation of Properties for Dynamic Verification

4.1.1.1 General Purpose Systems

In most of cases, the formal specification of the requirements that are to be dynamically verified is based on Linear Temporal Logic (LTL) [200] and variations of it including past and future time LTL (ptLTL and ftLTL respectively). Past and future time Linear Temporal Logics are modal logics for specifying properties of concurrent reactive systems and are used for analysing traces of execution of such systems. ptLTL provides temporal operators that refer to the past states of an execution trace, while ftLTL provides temporal operators that refer to the future/remaining part of an execution trace. In particular, the Temporal Rover (TR) tool [88] supports a future and past time Metric Temporal Logic (MTL). MTL [57] extends LTL with relative time and real time constraints. All four LTL future time operators can be constrained by relative time and real time constraints specifying the duration of the temporal operator. MTL constraints can specify lower bounds, upper bounds, and ranges for relative time and real time constraints.

In the context of monitoring oriented programming (MoP), any monitoring formalism can be added to the system. ptLTL, ftLTL and extended regular expressions (ERE), which can express patterns in strings in a compact way [218], have been used to formalise properties to be monitored [59]. The proposed algorithms use binary transition tree finite state machines (BTT-FSMs) to monitor ftLTL properties [59], as well as, formulas written in a logic based on EREs [218].

Havelund et al. [116, 117, 118] have developed several algorithms, which are relative to temporal logic generation and monitoring. For instance, they propose algorithms for past time logic generation by using dynamic programming [118]. Also they have used the MAUDE rewriting engine [63], for monitoring future time logic [116, 117] and have proposed algorithms that generate Büchi automata adapted to finite trace LTL [101].

Other logics/languages used for formalising properties are EAGLE [32] and HAWK [78]. EAGLE is a ruled-based language, which essentially extends the μ-calculus with data parameterization and past time logic. HAWK can be viewed as a specialization of EAGLE for JAVA, as it supports data binding and object reasoning. HAWK further extends EAGLE with event expressions, where events are restricted to method calls and returns. The integration of programming and logic as well as the notation and semantics of event expressions are similar to those used in modal logics like the π-calculus. HAWK also supports extended regular expressions.

According to the concept of Design by Contract (DBC) technique, introduced by Meyer [178] as a built-in feature of the Eiffel programming language, specifications of pre-conditions and post-conditions can be associated with a class in the form of assertions and invariants and subsequently be compiled into runtime checks. Jass [184] and jContractor [4] are two Java-based DBC systems. Jass is a pre-compiler, which turns the assertion comments into Java code. The JASS sub-language for specifying trace-assertions is similar to CSP [123], and its syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts, consisting of pre/post-conditions and invariants, with any Java class or interface.

The Monitoring and Checking (MaC) framework [157] is based on a logic that combines a form of past time LTL and models real-time via explicit clock variables. JAVA MAC [142], a prototype implementation of the MaC framework for monitoring and controlling applications written in Java, defines an event-based language to describe monitors. Note that, in the context of the Java MaC framework, events refer to information that holds instantly during the system runtime, while conditions are defined to illustrate information that holds for a time period. The Java MaC framework is composed of two specification event-based languages: the Primitive Event Definition Language (PEDL) and the Meta Event Definition Language (MEDL). PEDL is used for writing low-level specifications and is tightly related to the programming language. As such it deals with primitive events and conditions that might occur during the program execution, which are defined using program entities such as variables and methods. The operations on events and conditions can be used to construct more complex events and conditions from the primitive ones. A MEDL specification then makes use of these primitive events and conditions in order to state high-level requirements. Using MEDL, a user can specify the correctness requirements declaratively, without worrying about operational issues related to the monitor. The MaC framework also supports the declaration of variables of primitive types which can be updated by user-defined assignment statements upon arrival of new events. These variables can be referred to in formulas.

Recently, Mahbub and Spanoudakis [166] have developed a framework for monitoring the behaviour of service centric systems which expresses the requirements to be verified against this behaviour in event calculus [219]. In this framework, event calculus is used to specify formulas describing behavioural and quality properties of service centric systems, which are either extracted automatically from the coordination process of such systems (this process is expressed in WS-BPEL) or are provided by the user.

In the area of component based programming Barnett and Schulte [30] have proposed a framework which uses executable interface specifications and a monitor to check for behavioural equivalence between a component and its interface specification. In this framework, there is no need for recompiling, re-linking, or any sort of invasive instrumentation at all, due to the fact that a proxy module is used for event emission. The component's interface specifications are written in the Abstract State Machine Language (AsmL) [111], which is based on Abstract State Machines (ASM) [110]. This language is executable and supports non-deterministic specifications. Having native COM connectivity, one can not only specify and simulate components in AsmL but also substitute low-level implementations by high-level specifications. Specifications written in AsmL are operational specifications of the behaviour expected of any implementation. They provide a minimal model by constraining implementations as little as possible.

Robinson [204] has proposed a framework for requirements monitoring based on code instrumentation in which the high-level requirements to be monitored are expressed in KAOS. KAOS [79] is a framework for goal oriented requirements specification which is based on temporal logic. The KAOS modelling language can support all the phases of requirements acquisition and modelling, starting from initial functional and non-functional goals, formalising the meaning of such goals using temporal logic formulas and assigning the responsibility for the achievement of these goals to potential agents which may signify the system in question, systems that interoperate with it, and human actors interacting with the system. KAOS has also been used by Feather et al [96] in a framework that they have developed to monitor system requirements at runtime and which incorporates some capabilities regarding the reconciliation of requirements with the runtime system behaviour.

4.1.1.2 Security Oriented Systems

Some of the logics and languages reviewed in the previous section have also been used either as they were initially proposed or with some semantic modifications and extensions for the formalisation of security properties. Naldurg et al [187], for instance, have proposed a framework for intrusion detection which takes advantage of the capabilities of the EAGLE language for specifying the attack-safe behaviour of a system. EAGLE is suitable for expressing temporal patterns that involve reasoning about the data values observed in individual events and thus it allows the description of attacks whose signatures appear to have statistical properties, e.g., password guessing or denial of service attacks. For such attacks there is no clear distinction between an intrusion and a normal behaviour and the detection of intrusions involves collecting statistics during runtime and using them to evaluate the probability of the occurrence of an attack.

In the area of intrusion detection, Ko et al [148] have proposed a specification-based approach, which uses dynamic verification techniques to detect exploitations of vulnerabilities in security-critical programs. According to this framework, one has to specify a *trace policy* which describes the intended behaviour of programs with regards to security properties. A trace policy determines security-valid operation sequences of the execution of one or more programs. For specifying such trace policies, Ko et al. [148] have developed a grammar, called "parallel environment grammar (PE-grammar)" whose alphabet consists of system operations. A PE-grammar can express various classes of security trace policies, including behaviour related to access to system objects, synchronization, and operation sequencing and race conditions in concurrent or distributed programs.

Schneider [215] has developed a system called *Execution Monitoring (EM)* which can monitor violations of security policies by monitoring the execution steps of a system. This system is based on the security automata of Alpern and Schneider [8], which are a special type of Büchi automata. EM also incorporates mechanisms that can terminate the system execution if it is about to violate its security policy. Following the same automata-based formalism, Ligatti et al [160] extended the control capabilities of security automata by proposing edit automata, which can remove and add letters (i.e., system actions) to the words (i.e. execution traces) they recognise.

Having proposed a security-policy enforcing model which follows the general dynamic verification approach, Bandara et al. [25] have specified a language based on Event Calculus to model the system behaviour and write security policy specifications. The form of EC, which is used in this work was presented in [209] and consists of: (i) a set of time points (that can be mapped to the non-negative integers), (ii) a set of properties that can vary over the lifetime of the system (fluents), and (iii) a set of event types. System operations and domain-independent rules for policy enforcement were specified in this approach using these constructs. According to Bandara et al. [25], one can use EC to express system-models containing a combination of authorisation, obligation and refrain policies.

Janicke et al [131] have proposed a security model that allows expressing dynamic access control policies, which can be either time or event-driven. A system's overall security policy can then be composed out of smaller policies which capture specific requirements and which can be verified individually. The advantage of the access control model used in this work is that it allows expressing both parallel and sequential composition. Janicke et al. [131] based their security model on Interval Temporal Logic (ITL), a flexible notation for both propositional and first order reasoning about intervals of time. ITL allows to express properties for safety, liveness and timeliness. The policy model of Janicke et al. [131] provides a wide range of operators, for example to allow the dynamic addition/deletion of rules or to select different sub-policies based on to the occurrence of an event or a time-out. An important reason of choosing ITL was the availability of an executable subset of the logic, known as Tempura [186]. The use of ITL, together with its subset of Tempura, offers the benefits of traditional proof methods with the speed and convenience of computer-based testing through execution and simulation.

Brisset [44] has worked on establishing and ensuring the correct operation of a Java platform security mechanism for runtime authorization of un trusted applications in remote hosts. The resulting Java security mechanism, which is called SecurityManager and belongs to the JAVA runtime library, essentially embodies the security policy of the virtual machine. The verification technique used a CTL-based language, which extends CTL with JVM-specific atomic propositions. Thus, JVM-specific atomic formulas can be used for runtime authorization of untrusted applications. In order to verify an application against these formulas its byte-code is translated into pre/post-condition generators for CTL formulas on-the-fly.

Sekar et al. [217] presented an approach called model-carrying code (MCC) for mobile code security. The main components of MCC are: (a) a policy language for specifying security policies and a compiler for this language, (b) a language for specifying program behaviour models and techniques for extracting them, and (c) a policy refinement component which is based on model-checking techniques. Their language for policies and behaviour models is called Behavior Monitoring Specification Language (BMSL) and it is compiled into extended finite state automata (EFSA). EFSA are standard finite state automata (FSA) augmented with the ability to store values in a fixed number of state variables. These state variables are capable of storing values over both finite and infinite domains. The state of the EFSA is then characterized by its control state (which has the same meaning as the notion of state in the case of FSA), plus the values of these state variables. Each transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. For a transition to be taken, the associated event must occur and the enabling condition must hold. The assignments associated with the transition are performed when the transition is taken. In usual behavioural models the event alphabet of the EFSA consists of system-call names. On the other hand, security policies need to refer to particular uses of such system calls and be able to examine their respective arguments. These uses, for instance "read(sensitive_file)", augment the alphabet of EFSA with parameters to the initial system call names event alphabet. The resulting language is therefore able to distinguish the difference between the opening of a temporary file and the opening of a password file. Moreover, EFSA can also represent properties that refer to the arguments to system calls in the past, e.g. a program opens a file, whose name was given as an argument in the command line in the past.

For thoroughness we shall also mention certain higher-level languages and frameworks, which have been proposed for security requirements and policies. The KAOS framework, which we have already examined in the previous section on general-purpose formalisms, has been extended for modelling, specifying and analysing security requirements [153] by including the classical security concepts:

— *Adversaries/attackers* which are the malicious agents in the environment,

— *Threats* which are obstacles (anti-goals) intentionally set up by adversaries, and

— *Assets*, which must be protected against *threats*, are illustrated as passive or active objects.

The *Confidentiality*, *Integrity*, *Availability, Privacy, Authentication* and *Non-repudiation* requirements are sub-classes of the meta-class SecurityGoal in KAOS. Finally, the formal first-order, real-time, linear temporal logic of KAOS has been augmented with epistemic operators (Knows, Belief), which are needed in security-related properties (e.g. Authorized, UnderControl, Integrity or Using predicates).

Damianou et al. [77] have defined a declarative, object-oriented language, called Ponder, to specify security policies which can be monitored and applied at runtime. Ponder can be used to specify security policies regarding role-based access control to system resources, and general-purpose system management policies. Security policies are distinguished by Damianou et al. [77] in authorisation, obligation, refrain and delegation policies. Authorisation policies specify whether a subject is permitted to perform a particular action on a target; obligation policies specify management operations that must be performed when a particular event occurs and some supplementary guarding conditions are true; refrain policies allow system administrators to specify conditions under which certain operations should not be performed; and delegation policies specify which actions subjects are allowed to delegate to others. Ponder has been designed with the intention to be an extensible security policy specification language that would be able to cater for future types of policies and, rather than assuming a particular implementation platform, it could map to, and co-exist with, different underlying platforms.

In Service Oriented Computing, Baresi and Guinea [26] have proposed a framework for runtime monitoring of WS-BPEL processes. Monitoring rules are weaved at runtime into the process they must monitor and a proxy module supports their dynamic selection and execution [28]. Finally, they proposed a user-oriented language to integrate data acquisition and analysis into monitoring rules. Their monitoring rules define runtime constraints on WS-BPEL process executions and are expressed using the WSCoL language (Web Service Constraint Language). This development of this language has been inspired by the Java Modelling Language (JML) of Leavens, Baker and Ruby [154]. WS-CoL is a domain-independent policy assertion language for specifying user requirements (constraints) on the execution of Web services, which can be used within the framework of WS-Policy [214] and WS-Security [133]. WS-CoL is an assertion language augmented with features for allowing one to retrieve information that originates outside the process.

It distinguishes between *data collection* and *data analysis* to differentiate the phase in which information is collected (data collectionO, from the phase in which this data is analysed (data analysis). Data can be collected from the process directly (e.g., values of internal variable) but they can also come from external sources (e.g., exchanged SOAP messages). An example of a monitoring rule in this language could specify that all exchanged messages must be encrypted using the 3DES encryption algorithm.

4.1.1.3 Summary of specification languages for security and other system properties for dynamic verification

Table 4.1 gives a summary of various formal notations which have been used by different dynamic verification methods to express the properties to be verified and other functional and non functional characteristics of the systems and identifies languages and notations that have been specifically developed for expressing and verifying security properties.

As shown in the table most of the approaches deploy languages which are based on some form of temporal logic as these languages provide the necessary operators for expressing conditions about the temporal ordering and boundaries of occurrence of events which is required for the expression of most of the properties that need to be verified at runtime. The most popular formal notation for expressing security properties is Linear Temporal Logic (LTL) or extensions of it and languages with similar expressive power such as Event Calculus.

Languages for expressing security properties for dynamic verification	**Languages for expressing all types of properties for dynamic verification**
Behaviour Monitoring Specification Language (BMSL) and Extended finite state automata (EFSA)	EAGLE and HAWK
EAGLE	CSP-like specification
PE Grammar	LTL and its extensions
ITL	PEDL and MEDL
CTL (extended)	AsmL
Security automata	Event Calculus
Ponder	Ponder
KAOS	KAOS

Table 4.1 – Summary of formal languages used for dynamic verification

Some dynamic verification techniques reason about systems at both low and high level of abstraction, such as Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) in Java Monitoring and Checking (JavaMaC) framework [157]. PEDL is used for writing low-level specifications and is tightly related to the programming language, while MEDL specification makes use of primitive events and conditions in order to state high-level requirements.

4.1.2. Methods for Capturing Events

In the second stage of the general runtime verification process, the goal is to apply techniques so as to capture the real behaviour of the system during its execution.

As shown in Figure 4.3, existing event emission methods can be divided into *code modifying* and *non code modifying ones*. Code modifying event emission methods require direct access to the source or binary code of a system in order to insert code statements that will generate the events of interest. Code instrumentation is an example of a code modifying event emission method in which event generation statements are inserted manually into the code of a system. Aspect Oriented Programming (AOP) has also been used to generate events (through the weaving of aspects into binary or source system code). AOP is a *code modifying event emission method*, which can be considered as a subcategory of code instrumentation. Monitoring Oriented Programming [59] and Design by Contract [178] are also code modifying event emission methods which can be regarded as subcategories of Aspect Oriented Programming [140].

Non code modifying event emission methods generate events without altering the code of a system. Such methods access, modify and/or take advantage of capabilities of the general computational environment in which a system is executed, in order to generate the events flow. Reflective middleware approaches [53, 54, 170], proxy-based architectures [30] and the use of application programming interfaces (APIs) [166, 19, 46] constitute examples of event emission methods which belong to this category.

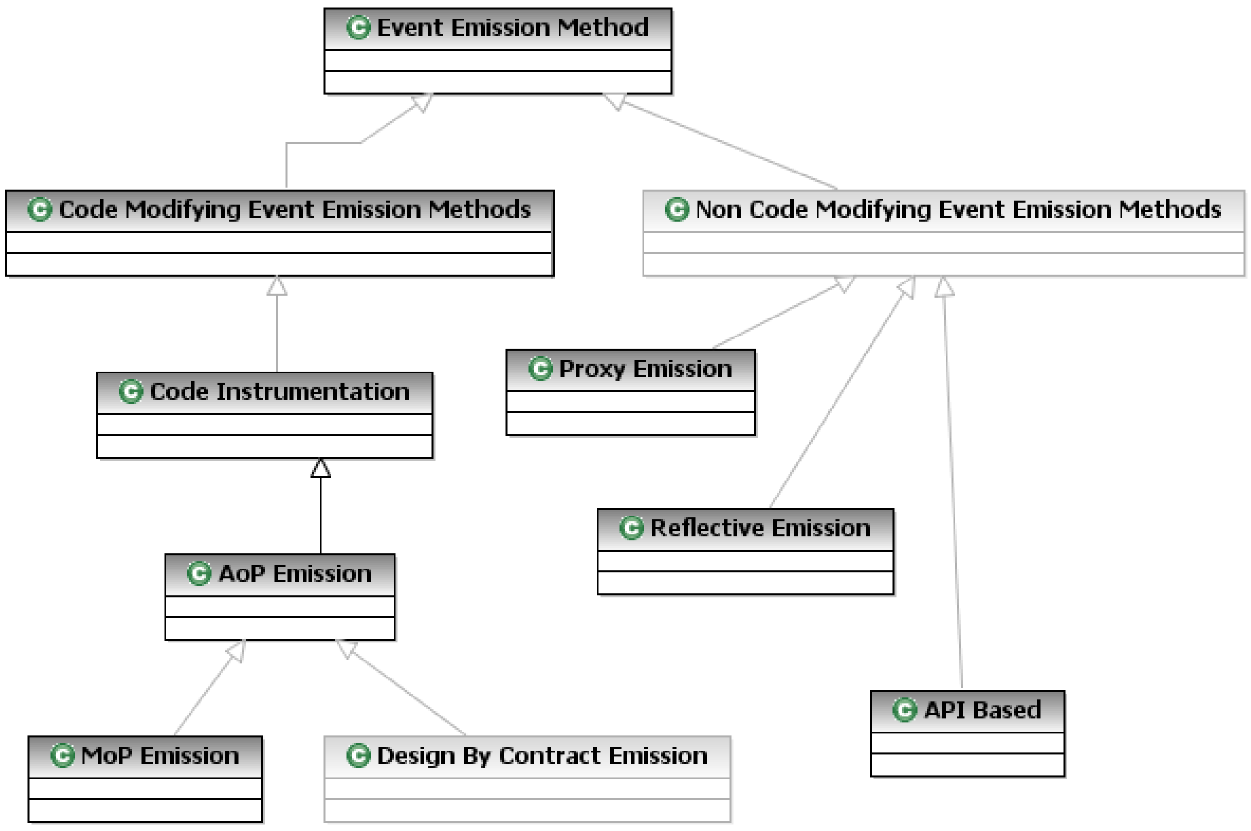


Figure 4.3 – Taxonomy of Event Emission Methods

4.1.2.1 Code-Modifying Event Capture Methods

4.1.2.1.1 Code Instrumentation

The technique of *code instrumentation* can be described [204] as the insertion of statements into the system's code (source or binary code) for monitoring purposes. Instrumentation can be done manually or automatically e.g. by using Jtrek-JSpy [102] or Joie [65] which automatically instrument Java byte code. During the execution of the instrumented code, an event stream is generated. The generated events can then be passed directly to external monitors or pre-processed before they reach the verification stage.

A tool using code instrumentation for capturing events in Java-based systems is RMon [204]. In Rmon, requirements are initially expressed in the KAOS framework [79], which provides a goal-oriented formal specification language based on temporal logic. Requirements are thus specified as high level goals which must be achieved by the system. These goals must then be mapped onto low-level events which can be monitored at runtime. The system's code is then instrumented in order to capture these low level events, using the Joie framework [65].

In the initial phase of the Java MaC architecture [142], low-level specifications (written in PEDL) are inserted into the byte code of the monitored program through an automatic instrumentation procedure. Furthermore, in the MONID tool [187] system-level events are generated by appropriately instrumented source code.

4.1.2.1.2 Aspect Oriented Programming

Aspect Oriented Programming (AOP) [140], also called Aspect Oriented Software Development (AOSD), was proposed to support the advanced identification, illustration and separation of non-functional concerns, which crosscut the system's main functionality. Complex programs include various crosscutting concerns (properties of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are scattered among different classes, complicating the development and maintenance of applications. As depicted in Figure 4.4, AOP enables the separation of crosscutting concerns during the development of the software. Specifically, the code implementing crosscutting concerns of the system, called aspects, is developed separately from other parts of the system. In AOP, locations in the program where aspect code can be woven, called *pointcuts*, are typically identified during development. Later, for example during compilation, an aspect weaver can be used to weave different aspects of the program together so as to form a program with new behaviour. AOP proponents argue that disentangling crosscutting concerns leads to simpler development, maintenance, and evolution of software [140]. Examples of AOP approaches include AspectJ [141] and Hyper/J [232].

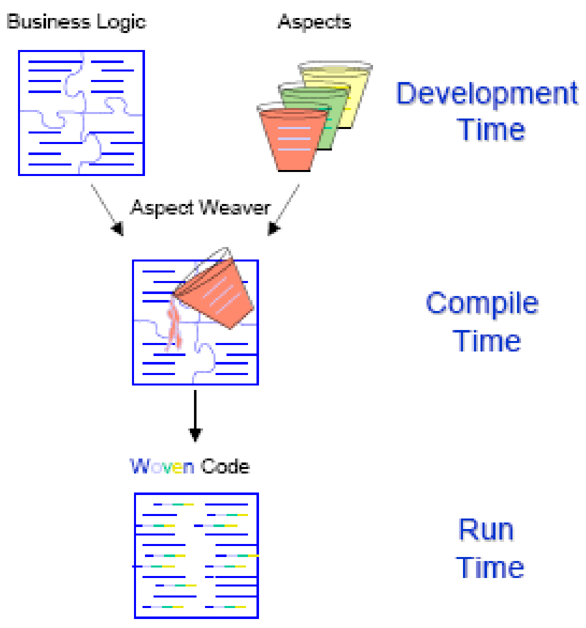


Figure 4.4 – Conceptual Representation of Aspect Weaving [140]

AOP supports dynamic re-composition in three major ways. First, most adaptations are relative to some crosscutting concern, such as quality-of-service or fault tolerance. AOP enables the code associated with these aspects to be written and managed independently of the application code as well as other parts of the system, such as traditional middleware platforms. Such a separation is needed in order to dynamically replace one instantiation of a particular solution for a concern with another. Second, although compile-time aspect weaving produces a tangled executable that cannot easily be reconfigured, delaying the weaving process until runtime provides a systematic way to realize dynamic re-composition [246, 122]. Finally, if adaptability itself is considered as a "generic" aspect [80, 253], then runtime weaving can be used to enhance the program with adaptive behaviour, not necessarily anticipated during the original development (e.g. to tolerate newly discovered faults or to detect and respond to new security attacks). This kind of upgrading is especially important in situations where the application is required to run continuously and cannot be easily halted for upgrade. However, the need of a formal aspect specification written in a domain-specific knowledge language or using logic, rather than the host programming language itself, is expressed in [59]. The mapping from specification to implementation, with the support of automatic code generation can then be formally verified.

In particular, AspectJ [141] provides an approach to implementing cross-cutting features in Java. AspectJ provides a pattern mechanism, called pointcuts, for capturing groups of events, called joinpoints, that may occur during a program's operation (such as method calls/receptions, constructor calls, field accesses, and exception events). The pattern matching mechanism includes regular expression matching, with wild-carding over fragments of method names, argument names, types etc. Extra code, called advices, can be associated with pointcuts, and is inserted by the AspectJ compiler into the join-points. Advices can inspect and modify data that are available at joinpoint events (e.g. method-call arguments and return values), and can create new data dynamically that is only shared with other advice.

For instance, Dingwall-Smith and Finkelstein [85] have developed an aspect oriented approach, in which system providers specify instrumentation code in separate classes, and define composition rules which determine how this code is to be merged with the application code, by using Hyper/J [232]. Also, Baresi and Guinea [27] have proposed a framework for runtime monitoring of WS-BPEL processes, in which monitoring rules are specified and weaved dynamically into the process they belong to. Furthermore, the instrumentation module of the JpaX framework performs a script-driven automated instrumentation of the program to be verified. The automated AOP environment package, which is used in JPaX [116, 119], is JSpy [102].

4.1.2.1.3 Design by Contract

Design by Contract (DBC), as proposed by Meyer [178] for the object-oriented language Eiffel, is a practical approach to runtime checking in applications. DBC is a lightweight formal technique, which allows one to add semantic information to a program by specifying assertions regarding the program's runtime state. Then, checks for specification violations are carried out at runtime. Such a technique stresses the importance of explicitly specifying the constraints that hold before (pre-conditions) and after a program is executed (post-conditions). The technique's name refers to a contract, which is made between the client and the supplier of a system module and defines conditions before and after the execution of the module. Thus, for monitoring reasons the entry and exit points of the module become the events that we want to observe.

In the context of the Eiffel object-oriented language, specifications of pre/post-conditions can be associated with a class in the form of assertions and invariants. Subsequently, inserted specifications can be compiled into monitoring code. In the Java language, there are two approaches which are based on DBC. Jass [33] is a pre-compiler which turns the assertion comments into Java code. Properties in Jass are called trace assertions and they specify permissible sequences of method calls in a CSP-like notation. Thus, processes, parallelism, conditionals and data exchange among processes can also be expressed. However, the trace assertions are interpreted loosely; no formal semantics is provided. The Jass pre-compiler translates the trace assertions into runtime checks.

4.1.2.1.4 Monitoring Oriented Programming

Monitoring Oriented Programming (MoP) is a paradigm which combines a formal specification with an implementation in order to form a system. In particular, it provides a light-weight formal method for runtime specification checks against the behaviour of the implementation. By using MoP, logical statements can be inserted anywhere in the program. These statements are simply Boolean expressions which can refer to past and future states of the program. A MoP user can insert such statements for different reasons e.g. to guide the system's execution, terminate the program or throw exceptions. Thus, MoP can increase the dependability of a system by monitoring its requirements at runtime and controlling it at the same time.

In particular, the statements, which can be inserted as annotations into the code, can be divided into three parts. The first part consists of a keyword defining the logic in which the rest of the inserted statements are expressed in. The second part comprises the definitions of the predicates and the formula to be monitored. Finally, user defined code which will be executed in case the monitored formula is violated is included in the third part, called a violation handler.

The general MoP paradigm is language and specification formalism independent. According to Chen and Rosu [59], a MoP environment should provide the capability of adding any logic framework on top of any target programming language via logic plug-ins, which can be publicly accessed. A logic plug-in consists of two modules, namely the logic engine and the target language shell. Logic engines translate formulae into monitors, encoded in an abstract representation (pseudocode). Then the language shell transforms the monitor pseudocode into the target language code. Thus, the logic plug-in can be considered as the code generator of the monitor.

Moreover, a MoP environment allows users to specify whether the monitoring code will be executed using the resources of the monitored program (internal monitor) or within a different process (external monitor). In the first case, the inserted logic statements contain the monitor's specifications which are replaced by the generated monitoring code in the end. Note that internal monitors, in general, cannot check for program deadlocks and unexpected terminations. In case that a monitor is executed as a different process, the inserted statements are replaced by instrumentation code which operates as an event generator. The user can specify whether the monitor should be executed synchronously or asynchronously with the monitored system and whether it should be executed on the same machine with the system or a different one.

4.1.2.2 Non Code Modifying Event Capture Methods

4.1.2.2.1 Reflective Middleware

Middleware technologies [94] have been designed to support the development of distributed systems. Their success has been mainly due to their ability of making distribution transparent to both users and software engineers, so that systems appear as single integrated computing facilities. However, hiding the implementation details from the application completely is very difficult in a mobile setting and not even always desirable, since mobile systems need to quickly detect and adapt to changes in their environment. A new form of awareness is needed to allow application designers to inspect the execution context and adapt the behaviour of the middleware accordingly.

Reflection and metadata can be successfully exploited to develop middleware targeted to mobile settings. By using metadata, we separate the middleware in two parts: what the middleware does and how the middleware does it. Reflection allows applications to inspect and adapt their metadata. In this way, applications can influence the way their middleware behaves, according to their current context of execution.

Capra, Emmerich and Mascolo [54] proposed a framework designed to ease the adaptation of applications to changing execution conditions. The model considers different layers (operating system, middleware, application, and user), each of which is described using metadata in order to ease their interaction. When the application invokes a service, the middleware uses both the application metadata and the metadata reflecting the current execution conditions to decide how to offer the requested service. Applications can also ask the middleware to be notified when specific execution conditions occur. This system allows for a fine adaptation of applications, but it requires that service calls be coded explicitly in the applications. However, a complete transparency is not possible if adaptation (which requires awareness) is desired.

XMIDDLE [170] is a middleware for mobile computing that focuses on the synchronization of replicated XML documents. In order to enable application-driven conflict detection and resolution, XMIDDLE supports the specification of conflict resolution policies through meta-data definitions using an XML schema.

CARISMA [54] is a context-awareness based reflective middleware. It includes a reflective API, which allows applications to dynamically inspect their current configuration and alter it to best suit the current environment. CARISMA maintains a representation of the execution context by interacting with the underlying network operating system. Based upon this representation, the application may behave in different ways. For example, an application attempting to send messages in low bandwidth availability may compress messages before emitting them, whereas it would send them uncompressed when bandwidth availability is high. The behaviour of the middleware with respect to the application is referred to as an application profile. There are two main aspects of an application profile, services and policies. Services describe the services offered to the application and which the middleware can customize. Policies describe the different variations in which the services can be delivered. In the prior example, the service the application is using is sending messages, and the different policies to deliver the service are sending either compressed or uncompressed messages based upon the context environment (high or low bandwidth). In CARISMA, each time a service is invoked, the middleware examines the application profile. Based upon the context of the application, the middleware determines which policy is best suited for the current context. This relieves the application of the burden of determining how to optimise its own behaviour.

4.1.2.2.2 Proxy Architecture

A proxy module acts as an intermediate between the monitored system and its environment, capturing their interaction and emitting the corresponding events. Thus, there is no need for code recompiling, re-linking or any other sort of invasive instrumentation at all.

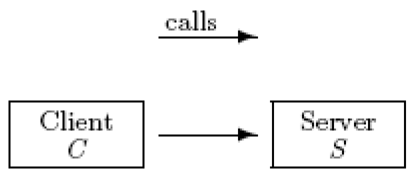


Figure 4.5 – A client-server architecture [30]

For component based programming, Barnett and Schulte [30] have proposed a framework which uses executable interface specifications and a monitor to check for behavioural equivalence between a component and its interface specification. Let us assume that a client–server architecture is used, like the one illustrated in Figure 4.5.

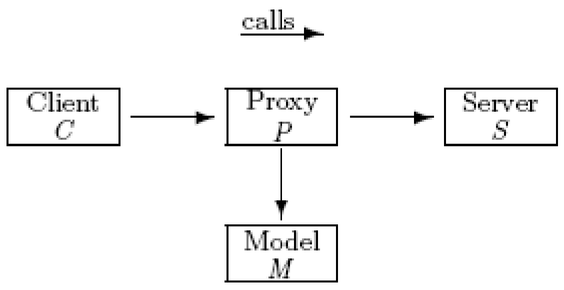


Figure 4.6 – Proxy Architecture [30]

A component, P, which essentially operates as a proxy, is inserted between the client C and the server S as shown in Figure 4.6. Using a proxy allows the interaction of the client C and the server S to be observed without having to modify either component. P can be created automatically from the definition of the interfaces, which C and S use in order to interact. The proxy forks all of the calls made from C to S so that they are delivered to both S and the (AsmL specification based) model, M, managing the concurrent execution of M and S. Then P compares the results from components M and S. P checks at each interface whether the results agree in terms of their success/failure codes as well as any return values. As long as, the results are the same, they are sent to C. In any other case, S and M are deemed not to be behaviourally equivalent.

4.1.2.2.3 *API-Based Event Capturing*

In the last non code modifying event emission subcategory, one finds approaches which make use of specific APIs for capturing and emitting events.

For instance, the Jnuke tool takes advantage of its virtual machine's (VM) specific API in order to observe the runtime behaviour of the monitored system. In particular, the event-based runtime verification API of JNuke's VM serves as a platform for various runtime algorithms. This API provides access to events occurring during program execution. Event listeners can then query the VM for detailed data about its internal state and thus implement any runtime verification algorithm, including detection of high-level data races [18] or stale-value errors [17] - see 4.3.7 for more details.

In the same family of event capturing methods is the prototype implementation of the specification based intrusion detection system, proposed by Ko et al. [148], which takes advantage of audit trails provided by the operating system. The prototype runs under the Solaris 2.4 operating system and uses the auditing services of the Sun BSM audit subsystem. The BSM audit subsystem provides a log of the activities that occur in the system. A BSM audit record contains information such as the process ID and the user ID of the process involved, as well as, the path name and the permission mode of the files being accessed. However, it does not contain information about the program the process is running. Therefore, an audit record pre-processor is used to associate the program identification with each audit record. The audit record pre-processor actually filters audit records that are irrelevant to the monitoring system and translates the BSM audit records into the format required by the monitoring system.

4.1.3. Checking for Violations

The third stage of dynamic verification is concerned with the checks that a monitor carries out to identify whether the runtime behaviour of a system conforms to certain properties. According to the taxonomy of Figure 4.2, the monitors with the most advanced capabilities are the "OC–pre–S" monitors. This category describes monitors, which verify the system's correct behaviour based on events describing the system's state before the execution of some action. The check is carried out while the system is halted, waiting for the monitor's reply. Once the monitor assures that the monitored properties hold, it allows the system to continue with its normal execution. If however a violation is reported, the monitor can force the system to execute some other action so as to remedy the current violation.

4.1.3.1 Checks for Admission

A widely used type of runtime checks is the check for admission. In this check a monitor checks an incoming request/application for admission, before actually honouring/executing it. In the following we shall examine some of the solutions for performing admission checks.

4.1.3.1.1 Proof Carrying Code

Proof Carrying Code (PCC) [191] can be used to increase security in systems executing untrusted, mobile code. With PCC, a program is supplied along with a proof of its correctness and this proof is in a form which can be easily verified mechanically before the program's execution. Therefore, it is now the code producer's responsibility to formally prove that the program will assure the safety properties specified by the code consumer, honouring the security policy of the underlying platform/system. Then, both the code and its proof are sent to the code consumer, where the safety properties are verified. A safety predicate is also generated directly from the native code to ensure that the accompanying proof does in fact correspond to the code sent. Once verified, the code can execute without any further checking. Any attempts to tamper with either the code or the safety proof result in a verification error.

The PCC binary life-cycle includes three stages:

- *Certification*: During this stage, the code producer compiles and generates a proof for the code, proving that the source program adheres to the safety policy of the code consumer. The proof can be produced by theorem proving.
- *Verification*: This stage is performed in the code consumer side. The code consumer verifies the proof part of the PCC binary code. The verification is performed by a simple algorithm, which is trusted by the consumer.
- *Execution*: The code consumer can execute the code without any further run-time checks.

For expressing safety policies Necula [191] has used first-order predicate logic, extended with predicates for type-safety and memory-safety. The untrusted code is in the form of machine code. For relating machine code to specifications they used a form of Floyd's verification-condition generator. Such a generator extracts the safety properties of a machine code program as a predicate in first-order logic. This predicate must then be proved by the code producer using axioms and inference rules supplied by the code consumer as part of the safety policy. For generating the safety proof, a theorem prover can be used, in the code producer's side.

Proof encoding can adequately be expressed using the Edinburgh Logical Framework (LF). LF is general and can easily encode a wide variety of logics, including higher-order logics. Another compact representation of proofs is a form of oracles, which guide a simple non-deterministic theorem prover in verifying the existence of the proof. For validating proofs encoded in LF, an LF type checker can be used. A non-deterministic logic interpreter can be used in the case that a proof is encoded as an oracle.

Initial research has demonstrated the applicability of PCC for fine-grained memory safety and shown the potential of it for other types of safety policies, such as controlling resource use.

PCC is based on principles from logic, type theory, and formal verification. There are, however, some potentially difficult problems to be solved before the approach is considered practical. These include a standard formalism for describing security policies, automated assistance for the generation of proofs and techniques for limiting the potentially large size of proofs that in theory can arise. In addition, the technique is tied to the hardware and operating environment of the code consumer, which may limit its applicability.

Comparing PCC to signed code, PCC is a prevention technique, while code signing is an authentication and identification technique used to deter the execution of unsafe code. Furthermore, the proof is structured in such a way that simplifies its verification, since it must be carried out efficiently without using any external assistance.

4.1.3.1.2 *Signature Verification of Signed Code*

Another technique for protecting a system, which is allowed to host mobile code, is by signing code with a digital signature. Using digital signatures, one can confirm the authenticity of the code, its origin, and its integrity. Typically the code signer is either the code producer or a trusted entity that has reviewed the code. Especially in mobile agents systems, where an agent can operate on behalf of an end-user or organization [231], the signature of an agent is used as an indication of the authority under which the agent operates.

Code signing is tightly bound with public key cryptography, which relies on a pair of keys (private and public) associated with an entity. One key is kept private by the entity and the other is made publicly available. Digital signatures benefit greatly from the existence of a public key infrastructure (PKI), since certificates containing the identity of an entity and its public key (i.e., a public key certificate) can be readily located and verified. The code signer applies an irreversible hash function to the code. The result of this function is a unique message digest of the code, which the code producer encrypts with his private key, thus forming a digital signature of the code. Because the message digest is unique, and thus bound to the code, the resulting signature also serves as an integrity protection against any malicious code modifications. The produced signature and the public key certificate can then be sent along with the code to the code consumer. The code consumer can easily verify the source and authenticity of the code by using the same hash function and the appropriate decrypting mechanism, which the code producer used to sign the code. If the signature verification succeeds, the code consumer can execute the code.

Note that the meaning of a signature may be different depending on the policy associated with the signature scheme and the party who signs. For example, the code producer, either an individual or an organization, may use a digital signature to indicate who produced the code, but not to guarantee that the code will be executed without faults.

Microsoft's Authenticode [106], enables Java applets or Active X controls to be signed, ensuring consumers that the software has not been tampered with and that the identity of the code producer is verified. Moreover, JDK 1.1 introduced the capability to digitally sign Java byte code (at least byte code files placed in a Java archive, called a JAR file), which expanded more with Java 2 [172]. From a certificate authority perspective, VeriSign provided a solution which addressed signed code issues for specific Netscape objects [243].

4.1.3.1.3 *Model Carrying Code*

Model Carrying Code (MCC) is an approach for supporting the safe execution of untrusted mobile code [217]. The central idea of MCC is that the code producer sends the code along with a high-level model, which describes the code's security-relevant behaviour. It should be noted that the generated model has to be usable by all code consumers. The automated model generation is based on model extraction via machine learning from execution traces. In the consumer's side, the model is checked for compliance with the consumer's security policy. If the security policy is satisfied, the code can be executed. In case there are conflicts, the consumer's policy can be refined, taking into consideration the code's functionality. When the code is executed, runtime verification methods are used to guarantee that the consumer's (refined) policy is not violated by the code (Figure 4.7).

By these means, the model bridges the semantic gap between the low-level binary code and the high-level security policies of the consumer. Moreover, the code producer does not have to know the consumer's security policies (as in PCC). Assuming that a model can be much less complex than the corresponding program, it is feasible for a consumer to automatically determine whether a model conforms to his security policies.

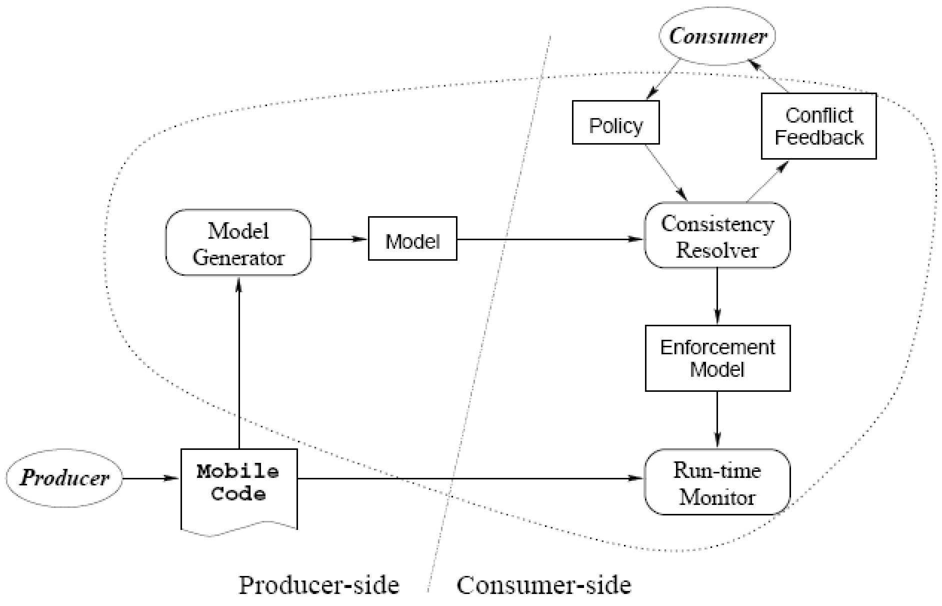


Figure 4.7 – The Model-Carrying Code framework [217]

4.1.3.1.4 Java Virtual Machine Byte-Code Verifier

The basic Java Virtual Machine (JVM) security model provides the capability of carrying out checks for admission for untrusted code, via a byte-code verifier [161]. In general the basic JVM security model comprises three related parts, namely the byte-code verifier, the class loader and the security manager. The JVM verifies all byte-code before execution.

The byte-code verifier reconstructs type information by inspecting the byte-code [255]. The types of all parameters of all byte-code instructions must be checked. The JVM specification lists what must be checked and what exceptions may result from a failed check. However, the JVM specification does not define when and how type verification should be done. Thus, while the process of verification in Java is defined to allow different implementations of the JVM, most Java implementations take a similar approach to verification. The most common verification process consists of byte-code checks on the class file itself and runtime checks, which confirm whether the referenced classes, fields and methods are existing and compatible to their attempted use.

The byte-code checks establish a basic level of security guarantees. In particular, the class file format is checked whether it is correct. This check is carried out with the class loader's cooperation. The code is also verified for the correct hierarchical structure of its classes. Thus, every class must have a super-class, final classes cannot have subclasses, final methods cannot be overridden and all field and method references in the constant pool (a heterogeneous array composed of five primitive types) must have legal names and classes. Moreover, the byte-code is verified by using data-flow analysis. By this means, it can be ensured that the operand stack can not be overflowed or underflowed, variables are properly initialised, register access is checked for using the proper value type, that method calls are done with the appropriate number and type of arguments, fields are updated with the appropriate type and all opcodes have the proper type of arguments on the stack and in the registers.

During the class execution runtime checks can occur, since some aspects of Java's type system cannot be statically checked, like dynamic linking. Java loads each class only when it is actually needed at runtime (dynamic linking). Thus, whenever an instruction calls a method, or modifies a field, the runtime checks ensure that the method or field exists, type-checks the call and checks that the executing method has the appropriate access privileges.

4.1.3.2 Post-Mortem Checks

Monitors which can only observe the runtime behaviour of a system ("O, pre, A" and "O, post, A") perform post-mortem checks. Post-mortem checks deal with properties which might not be of high importance. Proposed monitoring architectures for this category of monitors, like AMOS [64] and FLEA [95] maintain event logs and offer proprietary event pattern specification languages, or store events in relational databases and deploy standard SQL querying for detecting requirement violations [205].

4.2. Monitoring in Tuplespace-based Systems

Coordination models and languages aims to keep separate the description of the internal behaviour of the entities in a system from the description of their interaction.

In tuple-based (data-driven) coordination models client processes communicate and coordinate their activities by exchanging tuples of data via shared spaces (tuplespaces) [321].

The two key features of the tuple-based communication model are:

— time uncoupling: entities don't need to synchronize in order to communicate: once a tuple is written into the space it can be read at any later time (the lifetime of sender and receiver don't have to overlap).

— space uncoupling: entities don't need to know each other identity/address: tuple-based communication does not require sharing of naming conventions (independence from a shared addressing space) (tuples are independent of their producer). In tuple-based communication is associative in the sense that shared information is accessed by its attributes rather than names or address. Moreover a reader process needs to provide only a partial description (template) of the tuple to be retrieved.

The above features make the tuple-based coordination model appealing for Open Computing Environments, like the Internet and ad-hoc networks, where communicating entities can leave and join the network at will and more in general, where not all cooperating entities are known at design time.

Indeed in the context of Tuplespace-based Systems run-time monitoring has been proposed mainly as a strategy for:

— resource management;

— policy enforcement.

Before proceeding with the review of the approaches to run-time monitoring in the context of Tuplespace-based Systems the next paragraph introduce the key security issues for the e-based communication model and some of the solutions proposed in the literature. These approaches make extensive use of cryptographic techniques. The subsequent two paragraphs present the use of run-time monitoring for resource management and policy enforcement respectively.

Security issues in the tuple-based communication model

The key features that make tuplespaces an appealing coordination model become main security concerns in the context of Open Computing Environments. Since any entity is allowed to perform insertion, read and removal of tuples, a malicious or buggy entity can corrupt the integrity of data structures shared via a tuple space [262]:

— an entity can add an unbounded number of tuples (potential Denial of Service attack);

— an entity can remove all tuples matching a given template (potential corruption of data attack);

— it not possible to authenticate neither the producer nor the receiver of a tuple (potential disclosure of confidential information);

In recent years different proposal have been made to add security mechanisms to the original tuple-based communication model [263].

One of the first approaches proposed to enforce secure share of tuplespaces has been to join the use of multiple tuplespace (federated tuplespaces) with access control mechanisms such as password control. Following this approach multiple tuplespaces exist into the system each devoted to support communication between a groups of components (domain). In this kind of systems the security mechanisms in place have the granularity of the tuplespaces: in order to access the content of a tuplespace the client process must first get the access rights; once a client process gain access to a tuplespace it has full access on the content of the tuplespace.

The main limitation of password-controlled domains is that it assumes that tuples can be partitioned in domains. For some applications tuples must be visible in different domains, i.e. domain can overlap.

Moreover for some kind of applications its is required that some processes are able to access a tuple without accessing the value of the fields of the tuple: as an example a garbage collector process must remove garbage tuples without accessing the content of the filed since a buggy or malicious garbage collector could disclose confidential information.

Due to the limitations of the coarse-grain access control mechanism at the level of tuplespace more sophisticated access has been proposed in the literature.

In order to overcome the limitations of solutions dealing with the granularity of tuplespaces more recent works propose a data-driven approach to access control: the access to a tuple or one of its fields, is granted if the client process is able to proof it has knowledge of some data stored in the tuple.

In the SecSpaces model [262] the granularity of the access control is the tuple. Each tuple is annotated with a couple of control fields for each input operation (in and rd): partition key and asymmetric key.

SecSpaces extends the standard matching rule. The new matching rule uses the control fields for controlling the access to the tuples. In order to access a tuple a client process must provide, along with the template, a partition key and an asymmetric key. A tuple is returned if two additional matches occur with respect to the access operation:

- the partition key of the template matches the partition key of the tuple;
- the asymmetric key of the template corresponds to the co-key of the asymmetric key of the tuple.

The paper shows how the models can be used to support distributed session sharing and message brokering.

In SecOS [264] the granularity for the access control mechanisms is the fields: a client process can access a field if and only if it possesses the access key for that field. The SecOS model define two kind of keys:

- symmetric keys: that both lock and unlock fields;
- asymmetric keys: that belong to an asymmetric key pair (fields are locked with one key and unlocked with the other).

Indeed the SecOS model is quite different with respect to the standard tuplescapce model. First it the tuple model: tuple are unordered sequences of locked fields having the form (label:value). The pourpose of this redefinition is twofold. First labels are used to filter tuples during matching. Second labels are used as key to provide security in field access. redefines the standard matching rule for tuplespaces. According to the SecOS a tuple matches a template if each field of the template matches one field of the tuple. More specifically two fields match:

- If they have been locked with compatible keys. Symmetric keys are compatible if they are equals; asymmetric keys matches if they belong to the same key pair;
- If their values are equal or the template's value is the wildcard matching any value.

It has to be noted that the SecOS matching rule entails a concept of subsuption that is not present in the original tuplespace model.

The KLAVA system [261] is a middleware with cryptographic primitives to enable encryption and decryption of tuple fields. The work is inspired by SecOS but has different main concerns leading to some key difference:

- Conformance to original tuplespace model: hence KLAVA does not support subsumption as SecOS does;
- Code mobility: hence KLAVA imposes additional restriction on the coordination model.

In particular the code mobility explicit encryption and decryption and two stage pattern matching.

[263] propose a reference architecture for secure coordination of tuplespace client processes. The main components of the architecture are:

- a reference monitor: is a component intercepting all access to the tuples and verifying that each access is allowed by the security policy;
- security policy: is a set of rules specifying how information has to be accessed;

The paper focus on authentication and authorization requirements for tuple-based systems:

User of a tuple-space must authenticate to a protocol authentication process (PAP)

Other approaches propose to follow a strategy based on run-time monitoring.

4.2.1. Monitoring for resource management

4.2.1.1 LIGIA

The LIGIA system [284] uses run-time monitoring to manage garbage collection of tuplespaces. The LIGIA system allows client processes to create at run time new tuplespaces. In order to decide whether a given tuplespace is still required or not the system uses the concepts of process registration and tuple monitoring.

Process registration enables the LIGIA kernel to establish which process is executing a given instruction.

By means of tuple monitoring the system analyses the contents of the tuples being passed between registered processes. The analysis let the LIGIA kernel to keep up to date an internal graph structure representing the acquaintance relationship between processes and tuplespaces. The garbage collection of tuple spaces is driven by such a graph structure.

4.2.1.2 JavaSpaces Leases

The JavaSpace leasing mechanism give to JavaSpace compliant servers [280] a means to decide when discards tuples (named Entries in the context of the JavaSpace specifications).

When a client process writes a tuple into a space it must specify the amount of time for which the server should guarantee the tuple will reside into the tuplespace. The actual amount of leased time is established by the server according to available resources and returned to the client process as return parameter of a write operation.

JavaSpace compliant servers monitor their space in order to establish when discard tuples residing into the space.

A client process that wrote a tuple may renew or cancel the corresponding lease before lease expires.

4.2.2. Monitoring for policy enforcement

4.2.2.1 Law-Governed Interaction

The Moses toolkit [320] is a toolkit for developing tuplespace based systems. The toolkit has been applied to T-Space system and BinProlog implementation of the Linda coordination model.

The main objective of Moses is to enable enforcement of security policies by exploiting the concept of Law-Governed Interaction (LGI).

Within the Moses toolkit there is a controller monitoring the messages exchanged between components in the system. There are controllers both for client processes and tuplespace servers. Controller are placed between each components and the communication media. Each controller has the goal to enforce locally the laws of the coordination policy valid for the system.

A coordination policy is described by a four-tuple having the following components:

5. A set M of messages;

6. An group G of heterogeneous agents that can change their membership;
7. A mutable set of Control Sets (one for each agent in the group G); a control set is a collection of properties;
8. A law L; where L is a collection of rules having the form event->[op1,…,opn] where opk are primitive operations on control states and messages;

Each time a message is sent or received by the agent monitored by a controller, an event is raised within the controller itself and the controller evaluates the matching rules. The controllers evalute the rules sequentially in chronological order: the rules matching a new event are evaluated only when all rules matching the previous one have been evaluated.

4.2.2.2 JavaSpace Distributed Events

The JavaSpace Distributed Events enable JavaSpace clients to monitor at run time the content of a JavaSpace.

The JavaSpace technology incorporates the Jini Distributed Event model that makes possible to pass events across different Java Virtual Machines.

The JavaSpace Distributed Events model let a JavaSpace client process to register in a JavaSpace server its interest in the arrival of entries matching a specific template. When a new entry arrives the server notifies the event to all interested processes by remotely invoking a notify method of the registered process. It has to be noted that the invocation of the notify method is synchronous and hence during its execution neither other server activities nor the notify method of other clients are in execution.

4.2.2.3 LIME

The LIME [324] system extends the traditional tuplespace model to support the development of mobile applications. The LIME model aims to support both host mobility and code mobility.

The LIME coordination model introduces the concept of transiently shared tuple space. A transiently shared tuple space result form the aggregation of different tuplespaces. More in detail in the LIME model each client process has access to a permanently associated tuplespace (Interface Tuple Space) that is transferred along with the process across different hosts. The process access the content of the Tuple Space by means of the standard tuplespace operations (out, in and rd).

An Interface Tuple Space (ITS) makes visible to its process the content of all other ITSs located in the same host: each time a process move to a new host the content of its ITS is re-computed and the tuples residing on the ITS of the other co-located processes are made visible to the process. When a process move to a new host the content of all ITSs hosted in the same node is recomputed as well.

The ITS make visible also the content of the ITSs accessible across the network: each time a new host becomes visible in the network the ITS are synchronized. It is important to note here that in order to minimize data transfer the LIME model extends the standard tuplespace model with the notion of location: each time a process performs an out operation it specifies the intended location (i.e. ITS) for the produced tuple. The LIME kernel continuously monitors the network to detect if the intended ITS is in the reach: as soon as it becomes visible the tuple is transferred to the target ITS.

A part internal monitoring to support transiently tuple space the LIME system also has monitoring mechanisms to support reactive programming: client processes can register its interest in executing a non-reactive statement (reaction) as soon a tuple matching a given pattern is made visible through its ITS.

Each time a new tuple appears in the ITS where the reaction is registered the LIME kernel selects non-deterministically a matching reaction to execute. Once the execution of the reaction has completed the kernel goes on selecting and executing another matching reaction until no other matching reaction exists.

In order to avoid the need of a distributed transaction for each tuple space operation the LIME model offer also the possibility to register asynchronous reactions. An asynchronous reaction is not executed when a matching tuple is produced but in a later moment. The model guarantees that eventually the reaction will be executed.

4.2.2.4 Reactive Tuple Spaces

System relying on the concept of Reactive Tuple Spaces [265] aims to overcome the luck of flexibility of built-in associative mechanisms.

Although tuplespace coordination model offers advantages in term of both time and space uncoupling, the associative mechanism for tuple retrieval is fixed into the tuplespace server.

The drawback is that the coordination policies not directly supported by the tuplespace must be coded into the client process leading to break the separation of the description of the internal behaviour and coordination aspect.

In reactive tuple spaces models a tuple space is not just a repository of messages but indeed it is extended with the capacity to react to operations on the tuplespace: the tuple spaces can be programmed with the action to be undertaken in reaction of operations.

During their activity the tuplespace servers monitor the requests coming from client processes and apply the programmed reaction.

Systems relying on Reactive Tuple Spaces differs mainly on the data model and programming paradigm for the specification of reactions: in the MARS system [265] tuples are objects and reactions are programmed using an imperative language; in TuCSoN [287] tuples are standard tuples and reactions are programmed by means of a language based on first order logic.

4.3. General Purpose Dynamic Verification Tools

4.3.1. The Java PathExplorer (JPaX) framework

The Java PathExplorer (JPaX) is a tool for monitoring systems at their runtime [116, 119]. By using JPaX one can automatically instrument code and observe the system's runtime behavior. It can be used during development to provide more robust verification. It can also be used in an operational setting, to help optimize & maintain systems as they mature. Figure 4.8 illustrates an overview of JPaX architecture.

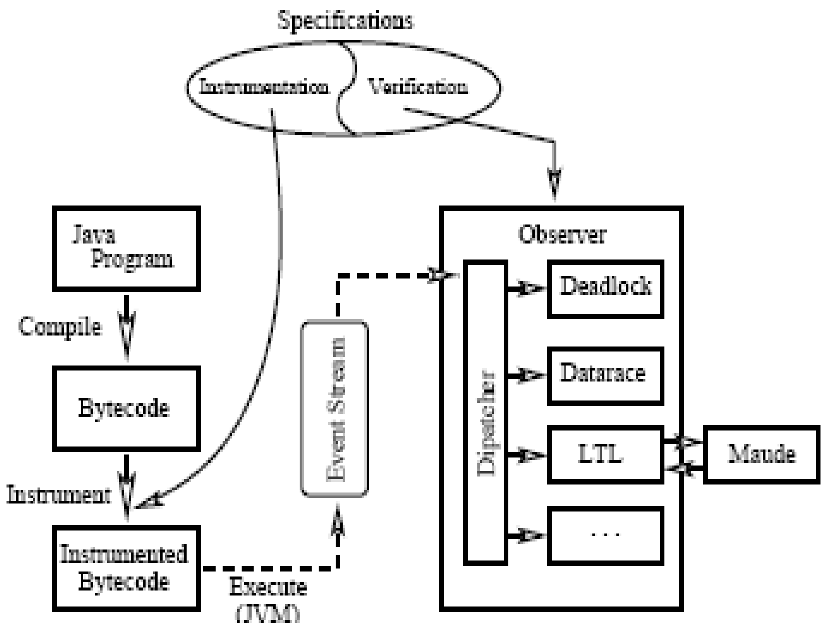


Figure 4.8 – The JPaX architecture [119]

JPaX consists of three modules:

1. Instrumentation module: It performs a script-driven automated instrumentation of the program to be verified, through which the byte-code is automatically instrumented.
2. Interconnection module: Its responsibility is to receive events about potential errors and transmit them to the observer module.
3. Observer module: It performs two kinds of verification:
    - Checks events against a user-provided requirement specification written in Maude, a formal, modularized specification and verification language. JPaX supports linear temporal logic (LTL), for both future and past time. Future time LTL uses execution traces as an underlying model making it convenient for program monitoring. Past time is useful for verification of safety properties.
    - Carries out error pattern analysis by exploring an execution trace and detecting potential problems such as error-prone programming techniques, e.g. locking practices that may lead to data races and/or deadlocks. The important and appealing capability of the error pattern analysis algorithms is that they can find potential errors, even in the case where errors do not explicitly occur in the examined execution trace. However, error pattern analysis may sometimes find errors which cannot exist. Two algorithms focusing on concurrency errors are implemented for JPaX:

i. The "Eraser" data race analysis algorithm. A data race occurs when two or more concurrent threads access a shared variable simultaneously without any locking mechanism and at least one thread intends to write in the variable. The "Eraser" keeps track of thread locks and variables to find data race conditions.

ii. Deadlock analysis algorithm. Deadlocks occur when multiple threads take locks in different order. For example, a deadlock condition occurs when:

 ! Thread A acquires Lock 1 while Thread B acquires Lock 2

 ! Thread A retains Lock 1 and asks for Lock 2 while Thread B retains Lock 2 and asks for Lock 1

 JPaX monitors locks during program execution to find potential deadlocks.

Using JPaX, a Java program byte-code is automatically instrumented using instructions from a user-provided script. This script defines what kind of error pattern detection algorithms should be activated and what kind of logic-based monitoring should be performed. The automated instrumentation tool, which is used in JPaX, is JSpy [102]. JSpy can be seen as an Aspect Oriented Programming tool in the sense that it is guided by rules, or aspects, which specify how a program should be transformed to achieve additional functionality. However, the main purpose of these aspects is to extract information from a running program. JSpy itself is built on top of the low-level JTrek instrumentation package [66].

As aforementioned, JPaX makes use of the Maude system [63]. Maude is a specification and verification system which supports rewriting logic. Rewriting logic is appropriate for expressing concurrent changes, which can naturally deal with state and with concurrent computations. Therefore, rewriting logic can be used like a universal logic, due to the fact that the syntax and operational semantics of other logics (such as temporal logics) can be expressed in rewriting logic.

The Maude rewriting engine can be used as:

- A monitoring engine during program execution. In JPaX, execution events are submitted to the Maude program that evaluates them against the requirements specification.
- Translation engine before execution. In JPaX, the specification is translated into a data structure optimised for program monitoring. This data structure is then used within the Java program to check the events at runtime.

JPaX produces either no output (when no errors are found) or a set of warnings. The warnings deal not only with runtime violations of high-level requirements written in temporal logic formulae but also with low-level error-detection procedures like concurrency related problems such as deadlock and data race algorithms.

The JPaX Java instrumentation module can be replaced with a C++ module to monitor C++ code. Experiments were conducted by the NASA Ames Robotic group on C++ code to check for deadlocks. JPaX located a potential deadlock that had not been previously detected during other testing [42].

To conclude, JPaX can also find potential errors, even in the case where errors do not explicitly occur in the examined execution trace. However, its logic-based monitoring adds an overhead to the normal execution of programs. Moreover, its error pattern runtime analysis can detect problems that do not really exist (called false positives).

4.3.2. *The Java Monitoring and Controlling Framework*

The Java Monitoring and Controlling (Java-MaC) framework uses formally specified properties to monitor Java programs at runtime [142]. Its architecture is shown in Figure 4.9. It can be divided in two main parts: the *static phase* (before a monitored entity runs) and the *runtime phase* (while the monitored entity is executing). During the static phase, the runtime modules, namely a *filter* (event generator), an *event recognizer* (event processing module), and a *run-time checker* (external monitor), are automatically generated from a formal requirements specification. During the runtime phase, events from the execution of the monitored program are collected and checked against the given requirements specifications.

The static phase of the Java-MaC architecture starts with a formal requirements specification, which is written in both high-level and low-level specifications. Java-MaC makes use of two event-based formal languages, the Primitive and the Meta Event Definition Languages (PEDL and MEDL), which are used for writing low and high level specifications respectively. PEDL is tightly related to the programming language. Specifications written in PEDL contain the definitions of primitive events and conditions expressed using these events. Such definitions are given in terms of program entities such as program variables and program methods and their purpose is to assign meanings to the program entities. MEDL specifications consist of required safety properties. Primitive events and conditions are used to express these safety properties. Intuitively, a condition is a state predicate and an event is an instantaneous state change. The reporting capabilities of the runtime checker are described in the MEDL specifications, as well. MEDL uses alarms to express a violation of a property. An alarm is an event that should not occur during an execution. If an alarm fires during an execution, then a user notification is issued.

Once the specifications are written, the next step is the generation of the runtime modules. Low-level specifications generate a filter that is inserted into the byte-code of the monitored program using an automatic instrumentation procedure. An event recognizer is also generated automatically by translating the PEDL specification. Similarly, a runtime checker is generated automatically from the higher-level MEDL specification.

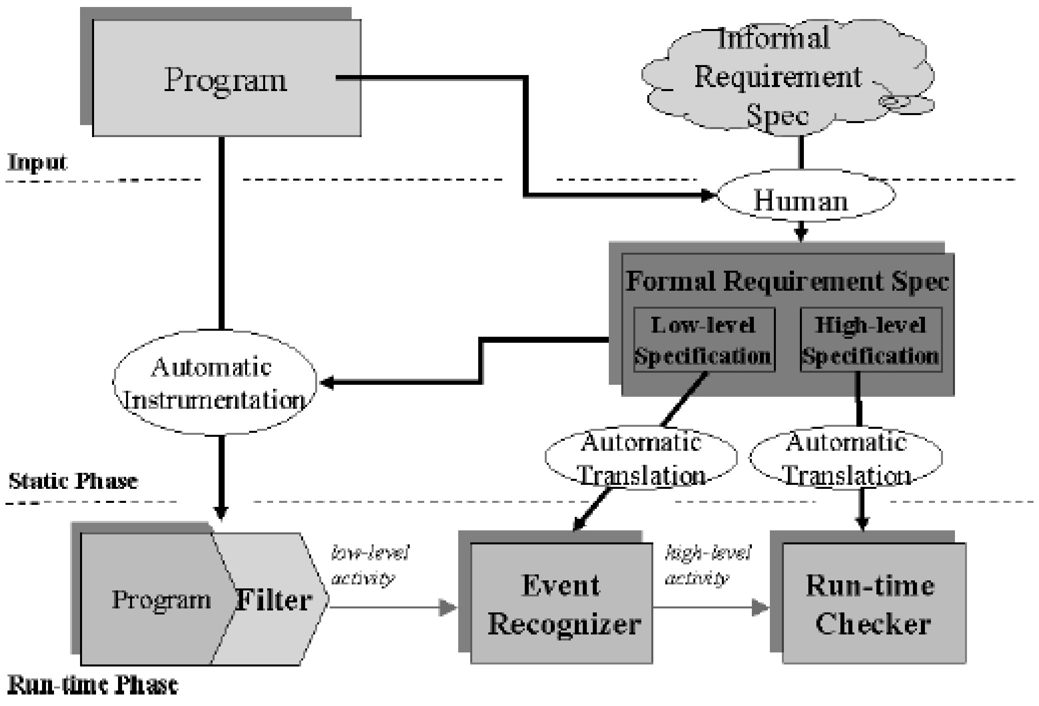


Figure 4.9 – The Java-MaC architecture [142]

During the runtime phase, the instrumented program is been monitored and checked against the requirements specification. Filter keep track of changes of monitored objects and send relevant information about the execution trace to the event recogniser. The event recognizer detects events from the state information received by the filter. An event can be either a primitive event (such as a method call) or a change in the state of a condition. Recognized events are sent to the run-time checker, which determines whether or not the current execution trace satisfies the requirements specification and raises an alarm if a violation is detected.

4.3.3. The Java Monitoring-Oriented Programming Framework

Chen and Rosu [60] proposed a development and analysis framework for Java, the Java Monitoring-Oriented Programming (Java-MoP). Java-MoP follows the MOP paradigm and thus monitoring is one of its fundamental principles. It also provides the capability of recovering from errors (specification violations) at runtime.

According to its proposed distributed architecture, annotations formally describing requirements on past, current and future states, have first to be inserted into the monitored Java source code, in the client side. Java annotation processors send these annotations to the appropriate logic plug-ins, which reside at the server side. Essentially, each of the logic plug-ins implements an algorithm for synthesising monitoring code for a specific formalism. Logic plug-ins, which have already been implemented, support past and future time variants of temporal logics, as well as, extended regular expressions. Furthermore, Jass [184] and JML [154] annotations can be used. These specific annotations do not require a special logic plug-in, only a Java shell to transform them into Java executable code.

Once the annotations have been transformed into Java executable code at the server side, they are sent to the client side. Java assertion processors integrate the received code in the system, according to the configuration attributes of the monitor. In addition, the client side modules are also responsible for the system's code instrumentation for emitting events, in case of an external monitor. In this Java-MoP implementation, AspectJ [141] is used as the instrumentation mechanism.

The checks, which can be carried out by using Java-MoP, depend on the monitoring properties. Thus, a monitor implemented in Java-MoP can check for class invariants at every change of class state or for method pre/post-conditions. Also, a monitor can be configured to halt the program's execution while it carries out specific checks which deal with critical properties (synchronised keyword). In case that a non-critical property must be checked, a monitor's reply may not be important, so the system keeps running during the check (asynchronised keyword).

MOP allows one to control the execution of a monitored program. By allowing users to specify handlers for the violation or validation of monitored properties, Java-MoP can support the runtime control and recovery of a monitored Java program. These handlers can either simply report errors and throw exceptions or take more complicated actions, like resetting states and performing alternative, error-correcting computations.

4.3.4. The Jassda Framework

An alternative to the Jass [33] approach, called Jassda [45], checks assertions on traces by observing the events generated for debuggers through the Java Debug Interface (JDI). An obvious shortcoming of this alternative is that the monitored program must be running in the debugging mode.

The Jassda tool allows the dynamic verification of a system written in Java against a CSP-like specification. The events from the monitored system are obtained through a general event extraction and dispatching facility, the Jassda framework [45]. This framework can also be used for other purposes, e.g., to log events or to stimulate a program for testing purposes.

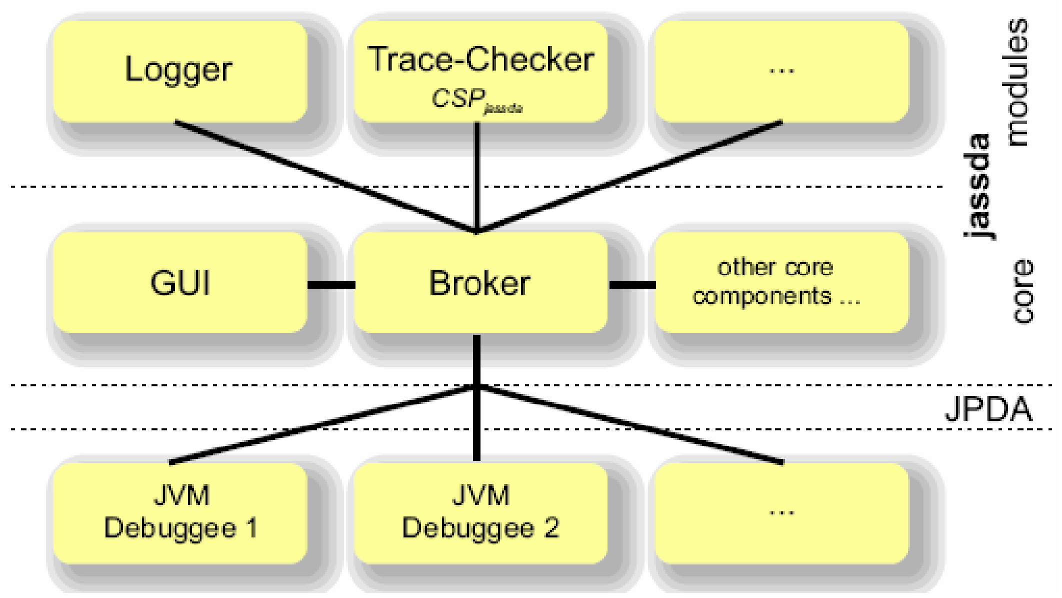


Figure 4.10 – The of Jassda architecture [45]

The architecture of the Jassda framework is shown in Figure 4.10. At the lowest level JVMs execute the monitored system's code (debuggees). These debuggees are connected to the Broker, which is the central component of the Jassda framework. The "Registry" database, an optional graphical user interface and the Broker build the Jassda core. Other Jassda modules connect to this core requesting and consuming events. The connection between the debuggees and the Jassda core transports the events that we want to observe. This connection is established by using the Java Platform Debugger Architecture (JPDA). The Jassda tool development aimed to achieve a method for monitoring Java programs which would be as less code intrusive as possible. The Java Debug Interface (JDI) [228] was used for this purpose.

During runtime the debuggees can be configured to generate events for several situations, e.g. a method has started or terminated, an exception has occurred, a breakpoint is reached, a class is loaded/unloaded, read/write access to a variable, a thread was started/stopped. After having emitted an event, the debugging VM can be configured to suspend execution and thus allow a deep view into the VM. For example, for each currently running thread its stack trace can be analyzed or for each class its inner structure (like super-classes and implemented interfaces) can be read. Even the byte-code of every method can be accessed for further analysis.

The Logger module logs the execution of a Java system by writing its sequence of events into a file. The amount of information that can be derived from an event as well as the alphabet of events can be configured by an XML-based configuration file. The most important event listening module is the Jassda Trace–Checker. The Trace–Checker reads one or more trace specifications written in CSPJassda and builds an internal process representation for the set of legal traces. With every received event the Trace–Checker will ensure that this actual sequence of events is a legal trace of the specification's process representation or stop the program and inform the user of the specification violation.

4.3.5. The Temporal Rover Toolset

The Temporal Rover [88] is a commercial toolset, which performs dynamic verification of temporal properties over programs written in Java, C, C++, VHDL, Verilog, and ADA. This is achieved by adding extra LTL/MTL assertions to the program source code. These assertions are embedded as comments into the source code. The Temporal Rover parser converts program files into new files, which are identical to the original files except for the assertions that are now implemented in source code.

The Temporal Rover adopts a coarse-grained view of the state model. A state constitutes the values of variables within the scope of a given method. Method execution is viewed as an event that causes transition between states, and properties are evaluated only at the completion of a method execution. Clearly, it misses invalid states that may occur during the execution of a method. Properties are written inside methods and predicates map to the variables within the scope of the method. Consequently, each property has a unique perspective of the environment that it is validating and properties may not be composed. For example, even though one would imagine that two contradicting properties could be composed and reduced to "false", this is not the case under the Temporal Rover's state model. A property's notion of time refers to the next execution of the method containing it. Two properties may therefore carry different semantics for the next-state operator. Another limitation of Temporal Rover is that under its state model one can not reason about control intensive properties such as method x must never execute after method y. The DBRover is a distributed-monitor version of the Temporal Rover where assertions are monitored on a remote machine, using HTTP, sockets or serial communication with the underlying target application.

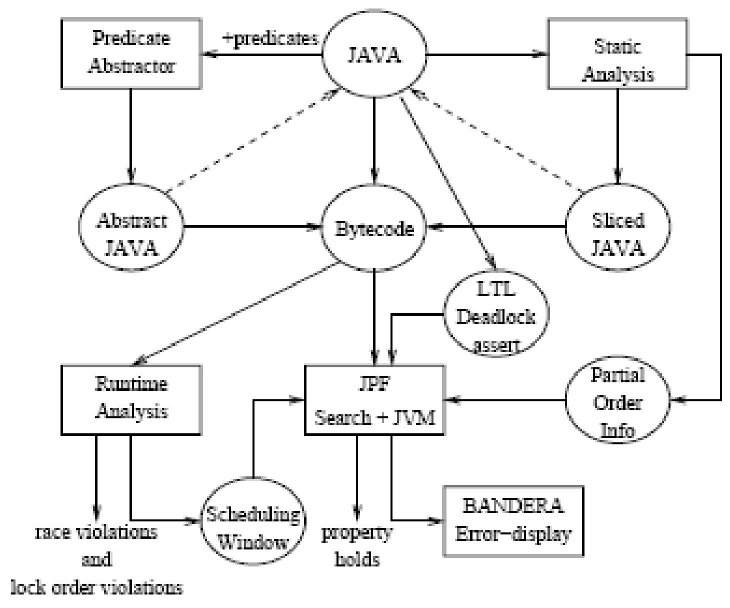
4.3.6. The Java PathFinder (JPF) Framework

Java PathFinder (JPF) [245] is a model checker for Java byte code. More specifically, it is a specialized Java Virtual Machine (JVMJPF), which runs on top of the underlying host JVM. In contrast to the standard JVM, JVMJPF executes the program in all possible ways. The state space of a program is thus the resulting computational tree, whose branches are determined by the threads' instructions and possible values of input data. JPF supports depth-first, breadth-first as well as heuristic search strategies to guide the model checker's search in cases where the state explosion problem is too severe [108]. JPF contains no mechanism of its own to specify user-defined properties, but rather integrates with the Bandera toolset [69] and accepts the Bandera Specification Language (BSL) [68]. Even though JPF carries an elaborate state model (being able to capture every state of the JVM), temporal property specification is limited to the capabilities of BSL. Figure 4.11 depicts the JPF architecture.

Like other model checkers for concurrent programs, JPF supports the partial order reduction (POR) [62]. The purpose of this technique is to lower the state space size via including in the state space only one interleaving of instructions that are both independent and executed by different threads. The consequence is that JPF actually traverses a reduced state space where each state is associated with one of the following events ("points") in the byte code execution:

(a) Scheduling point: The current instruction is thread scheduling relevant (e.g. it accesses a shared variable, starts/stops a thread, blocks a thread, etc.)

(b) Value point: A value selection takes place.



(Dotted lines indicate iterative analysis)

Figure 4.11 – The JPF Architecture [245]

In order to check a code unit (e.g. a method) for different values of input data, JPF contains a static class Verify which provides methods for a systematic selection of values of virtually any type. The methods of Verify are to be called in the checked code. For example, if the checked code unit executes Verify.random(3), an integer value from the range 0..3 is selected. However, after reaching an end state, JPF backtracks up to the Verify.random(3) call and selects another value from 0..3; this is repeated until all the values from this interval have been used for execution. By employing methods of Verify the state space size increases since each selected value creates a different branch in the state space.

By default, JPF searches the state space of the checked program for “low-level” properties like deadlocks, unhandled exceptions and failed assertions. However it is extensible via the publisher/listener pattern and as such it allows for observing more general properties. Since Java code assertions must always hold, temporal properties specified outside of BSL can be checked as well. This way, listeners can check for specific state-based properties.

Each state of a checked program in JPF consists of the heap, static area and stacks of all threads. When traversing the state space, JPF checks whether the current state has already been visited. If this is so, it backtracks to the nearest scheduling/value point, for which there exists an unexplored branch and continues along that. This backtracking is based on keeping a stack representing the currently explored path in the state space (an item in the stack determines the list of yet unvisited branches).

The Bandera toolset [114] is a collection of program analysis, transformation, and visualization components designed to allow experimentation with model-checking of Java programs. Bandera takes as input a Java source code and a program specification written in Bandera’s temporal

Specification Language (BSL), and produces a program model and a specification as input to model-checking applications, like Spin [124] and Java PathFinder [245]. Then, Bandera uses the corresponding model-checker to prove whether the model satisfies the required specification (i.e. whether the Java program satisfies the BSL specification). If the specification is not satisfied, then a counter example trace is returned. Bandera uses this to show the problematic execution path directly in the original Java code. Bandera deals with the state explosion problem and the fact that the program state models must be finite by providing data abstraction and program slicing methods when customizing the model. These features help produce a much smaller finite state model of the Java program.

In particular, Bandera consists of five major components:

— Property specification is supported in Bandera through the use of global properties (e.g., deadlock) and application specific properties (e.g., assertions and temporal logic formulas). Users define observations of the execution state of a Java program, as predicates over program locations and data values in the program. Assertions and temporal formulas are then defined in terms of those observations.

— Program slicer: Automates the elimination of program components that are irrelevant for the property under analysis. Slicing criteria are automatically extracted from the observable predicates that are referenced in the property. Bandera's Java slicer treats multi-threaded programs [115] and is based on calculation of the program's data dependence graph.

— Program abstraction which can be summarized as: (i) definition of an abstraction mapping, which is appropriate for the property being verified, (ii) use of the abstraction mapping to transform the temporal property into an abstract property, (iii) use of the abstraction mapping to transform the concrete program into an abstract program, (iv) checking whether the abstract program satisfies the abstract property, (v) reasoning about the satisfaction of the concrete property by the concrete program.

— Verification code generator: Transforms the sliced, abstracted program into the input format of a selected model checker. This component is also responsible for establishing the correspondence between the states of the produced model and the states of the original program.

— Counter-example interpreter: Involves the mapping of low-level, verifier-specific counter-examples back to the Java source code. Facilities for navigating through the counter-example and displaying the values of both stack and heap allocated data are provided through a debugger-like interface.

4.3.7. The JNuke tool

JNuke is a framework for static and dynamic analysis of Java programs [16, 19]. It was originally designed for dynamic analysis, including explicit-state software model checking and runtime verification.

JNuke's virtual machine (VM) is the core element of the framework. For generic runtime verification, the engine executes only the program once according to a given scheduling algorithm. The VM API allows for event-based runtime verification through various runtime algorithms. This API provides access to events occurring during the program execution. Event listeners can, then, query the VM for detailed data about its internal state and thus implement any runtime verification algorithm, including detection of high-level data races [18] and stale-value errors [17].

Before the execution of the monitored program, the class loader transforms the Java byte code into a simplified form containing only 27 instructions, which is then transformed into a register-based version [16]. Execution of the program generates an event trace. During execution, the runtime verification API allows event listeners to capture this event trace. These listeners are used to implement scheduling policies and runtime verification algorithms, like Eraser [213] and detection of high-level data races [18]. The verification algorithm is responsible to copy data it needs for later investigation, as the VM is not directly affected by the listeners and thus may choose to free data not used anymore. Figure 4.12 presents an overview of the JNuke VM and how a runtime verification algorithm can be executed by using callback functions in the VM.

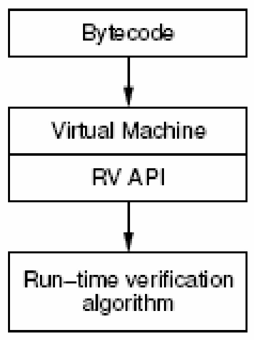


Figure 4.12 – Runtime verification in JNuke [16]

JNuke was expanded with static analysis capabilities at a later stage. Static analysis is usually faster than dynamic analysis but less precise, approximating the set of possible program states. In static analysis, iterations over these approximated states are carried out until a fix point of them is computed [72]. Properties checked with static analysis require summary information of dependent methods or modules. Figure 4.13 illustrates the separate classical approaches to dynamic and static analysis.

On the other hand, dynamic analysis examines properties against an event trace originating from a program execution. By using a free data flow analysis graph [182] static analysis can work similarly to the dynamic execution. Analysis algorithms based on such a graph can allow for non-deterministic control-flow and use sets of states rather than single states in its abstract interpretation [16]. Moreover, in such a graph data locality is improved because an entire path of computation is followed as long as valid new successor states are discovered. Thus, all Java methods can be executed, allowing for a generic analysis algorithm to be executed under both static and dynamic analyses. The chosen analysis algorithm runs until an abortion criterion is met or the full abstract state space is exhausted.

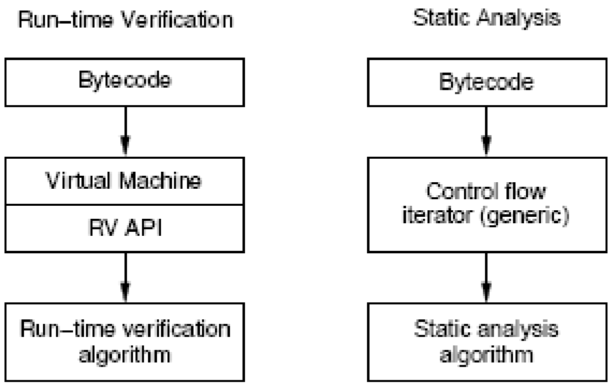

Run–time Verification
Bytecode
Virtual Machine
RV API
Run–time verification algorithm
Static Analysis
Bytecode
Control flow iterator (generic)
Static analysis algorithm

Figure 4.13 – : Classical approach to dynamic & static analysis [16]

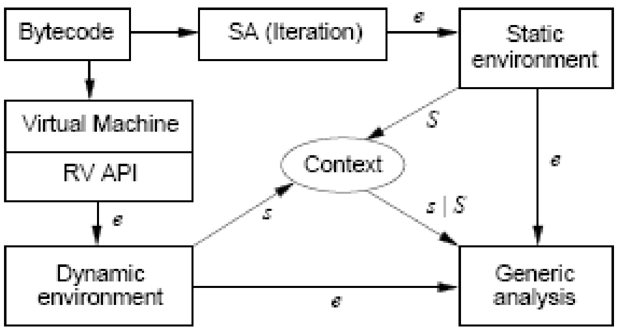


Figure 4.14 – Generic analysis for both a static & dynamic environment [16]

A generic analysis represents a single program state or a set of program states at a single program location. It also includes a number of event handlers, which model the semantics of byte code operations. Both static analysis and runtime analysis trigger an intermediate layer, which evaluates the events. The environment hides its actual nature (static or dynamic) from the generic algorithm and maintains a representation of the program state that is suitably detailed.

Figure 4.14 shows the generic analysis principle. Run-time verification is driven by a trace, a series of events e emitted by the runtime verification API. An event represents a method entry or exit, or execution of an instruction at location l. Runtime analysis examines these events directly. The dynamic environment, on one hand, uses the event information to maintain a context c of algorithm-specific data before relaying the event to the generic analysis. This context is used to maintain state information s that cannot be updated uniformly for the static and dynamic case. It is updated similarly by the static environment, which also receives events e, determining the successor states at location l which are to be computed. The key difference for the static environment is that it updates c with sets of states S. Sets of states are also stored in components used by the generic algorithm. Operations on states (such as comparisons) are performed through delegation to component members. Therefore the "true nature" of state components, whether they embody single concrete states or sets of abstract states, is transparent to the generic analysis algorithm, which can thus be used either statically or dynamically.

The abstract domain for the static analysis is chosen based on the features required by the generic analysis algorithm to evaluate given properties. Both the domain and the properties are implemented as an observer algorithm in JNuke. Future algorithms may include an interpreter for logics such as LTL. Interpretation of events with respect to temporal properties would then be encoded in the generic analysis while the event generation would be implemented by the static and dynamic environment, respectively.

4.3.8. Summary of Dynamic Verification Tools

The following table summarizes the surveyed verification tools in terms of the general dynamic verification approach steps of Figure 4.2.

Tool	**Language for Properties Formalization**	**Methods for Events Emission**	**Monitor**	**Category**
JPaX	Temporal logic in Maude rewriting tool	Automated Instrumentation using JSpy (modified JVM)	Observer	O, pre/post, A
Java-MaC	Past-time interval temporal logic	Automated Instrumentation (Instrumentor)	Runtime Checker	O, pre/post, A
JMoP	ptLTL, ftLTL, EREs	Automated instrumentation by using AspectJ	Embedded in code or parallel process to the system on the same or different machine	OC, pre/post, S/A
Jassda	$\mathrm{CSP}_{\mathrm{Jassda}}$	API based (from JVMs by using the Java Debug Interface)	Trace checker	OC, pre/post, A
Temporal Rover	LTL/MTL assertions	Intrumentation	Embedded (using alternating finite automata)	OC, pre/post, S

JPF	User defined assertions, LTL (by using BANDERA)	By using BANDERA's abstraction capability	JVM^{JPF}	OC, A
JNuke	-	API based (JNuke VM with RV API)	Runtime verification algorithm	O, post, A

Table 4.2 – Summary of Dynamic Verification Tools

4.4. Dynamic Verification Tools Focusing on Security Properties

4.4.1. Firewalls

A firewall [107] is a device or group of devices that controls access between networks. A firewall generally consists of filters and gateway(s), varying from firewall to firewall. It is a security gateway that controls access between the public Internet and an intranet and is a secure computer system placed between a trusted network and the generally not trusted Internet. A firewall is an agent, which monitors network traffic in some way, blocking traffic it believes to be inappropriate or dangerous. It is well known that Internet access provides benefits to individual users, government agencies and most organisations. But this access often creates a threat as a security flaw. The protective device that has been widely accepted is the firewall. When inserted between the private intranet and the public Internet it establishes a controlled link and erects an outer security wall or perimeter. The aim of this wall is to protect the intranet from Internet-based attacks and to provide a choke point where security can be imposed.

The main purpose of a firewall is to impose restrictions on packets entering or leaving the private network. All traffic from inside to outside, and vice versa, must pass through the firewall, but only authorised traffic will be allowed to pass. Packets are not allowed through unless they conform to a filtering specification.

Firewalls, as mentioned above, create choke points between an internal private network and the not trusted Internet. Once the choke points have been clearly established, the device can monitor, filter and verify all inbound and outbound traffic on the basis of IP source and destination addresses and TCP port number.

The firewall also enforces logging, and provides alarm capacities as well. By placing logging services at firewalls, security administrators can monitor all access to and from the Internet. Good logging strategies are one of the most effective tools for proper network security.

A firewall can limit network exposure by hiding the internal network systems and information from the public Internet.

The firewall certainly has some negative aspects: it cannot protect against internal threats such as a trusted entity which cooperates with an external attacker; it is also unable to protect against the transfer of virus-infected programs or files because it is impossible for it to scan all incoming files, e-mail and messages for viruses. However, since a firewall acts as a protocol endpoint, it may use an implementation methodology designed to minimise the likelihood of bugs.

Firewalls can be classified into three main categories: packet filters, circuit-level gateways and application-level gateways.

4.4.1.1 Packet Filters

Packet filters are one of several different types of firewalls that process network traffic on a packet-by-packet basis. A packet filter's main function is to filter traffic from a remote IP host, so a router is needed to connect the internal network to the Internet. Packet filters typically set up a list of rules that are sequentially read line by line. Filtering rules can be applied based on source and destination IP addresses or network addresses, and TCP or UDP ports. Packet filters are read and then treated on a rule-by-rule basis. A packet filter will provide two actions, forward or discard. If the action is in the forward process, the action takes place to route the packet as normal if all conditions within the rule are met. The discard action will block all packets if the conditions in the rule are not met. For example, if TELNET services were forbidden in a network protected by a packet filter, then the rule in the packet filter would discard any packet from the Internet with destination IP within the intranet and port 23. Thus, a packet filter is a device that inspects each packet for predefined content. Although it does not provide an error-correcting ability, it is almost always the first line of defence.

Since a packet filter can restrict all inbound traffic to a specific host, this restriction may prevent a hacker from being able to contact any other host within the internal network.

However, the significant weakness with packet filters is that they cannot discriminate between good and bad packets. Even if a packet passes all the rules and is routed to the destination, packet filters cannot tell whether the routed packet contains good or malicious data. Another weakness of packet filters is their susceptibility to spoofing where the attacker sends packets with an incorrect source address.

4.4.1.2 Circuit-level gateway

The circuit-level gateway represents a proxy server that statically defines what traffic will be forwarded. Circuit proxies always forward packets containing a given port number if that port number is permitted by the rule set. This gateway acts as an IP address translator between the Internet and the internal system. The main advantage of a proxy server is its ability to provide Network Address Translation (NAT). NAT hides the internal IP address from the Internet. NAT is the primary advantage of circuit-level gateways and provides security administrators with great flexibility when developing an address scheme internally.

Circuit-level gateways are based on the same principles as packet filter firewalls. When the internal system sends out a series of packets, these packets appear at the circuit-level gateway where they are checked against the predetermined rules set. If the packets do not violate any rules, the gateway sends out the same packets on behalf of the internal system.

The packets that appear on the Internet originate from the IP address of the gateway's external port, which is also the address that receives any replies. This process efficiently shields all internal information from the Internet.

4.4.1.3 Application-Level Gateways

The application-level gateway represents a proxy server, performing at the TCP/IP application level, that is set up and torn down in response to a client request, rather than existing on a static basis. Application proxies forward packets only when a connection has been established using some known protocol. When the connection closes, a firewall using application proxies rejects individual packets, even if the packets contain port numbers allowed by a rule set.

The application gateway analyses the entire message instead of individual packets when sending or receiving data. When an inside host initiates a TCP/IP connection, the application gateway receives the request and checks it against a set of rules or filters. The application gateway (or proxy server) will then initiate a TCP/IP connection with the remote server. The server will generate TCP/IP responses based on the request from the proxy server. The responses will be sent to the proxy server (application gateway) where the responses are again checked against the proxy server's filters. If the remote server's response is permitted, the proxy server will then forward the response to the inside host.

Application level gateway technology using proxy services has several advantages. Proxy services enforce high-level protocols such as HTTP and FTP. Information about the communications passing through the firewall server is maintained by the proxy service. Proxy services can permit access to certain network services, while denying access to others. Packet data can be processed and manipulated by proxy services. Internal IP addresses are shielded from the external world because proxy services do not allow direct communications between external server and internal computers. Administrators are able to monitor attempts to violate the firewall's security policies using the audit records that proxy services can generate.

Although application level gateways provide increased security over a packet filter there are some disadvantages to using an application level gateway. Application level gateways are slower since inbound data is processed by the application and by its proxy. A new proxy usually must be written for each protocol that is to pass through the firewall. This can cause the number of available network services and their scalability to be limited. Proxy services are vulnerable to operating system and application level bugs.

4.4.2. Intrusion Detection Systems

An Intrusion Detection System (IDS) is software designed to detect unauthorised access to a computer system or network. This may take the form of attacks by adversaries using automated tools, the attack tools, which are designed for violating the security policy of a system. An IDS is required to detect different types of malicious network traffic and computer utilization. This includes network attacks against vulnerable services, data driven attacks on applications, host based attacks such as unauthorised logins and access to sensitive files, and malware (viruses, trojan horses, and worms).

The primary goals of IDS are:

— Intrusion detection for known and unknown attacks. (In the latter case, a learning mechanism, for new types of attacks or for changes observed in system user activities, should be implemented in an IDS).

- Intrusion detection in admissible time limits.
- Precise results generation. (IDS results must be precise).

Other desirable characteristics of IDS are:

- To run continually
- To be fault tolerant
- To be configurable
- To be adaptable
- To be scalable
- To allow dynamic reconfiguration

A general model for IDS was proposed by Denning [81]. The main assumption of this model is that the exploitation of system vulnerabilities requires irregular usage of accepted commands, so that the security breaches can be monitored. The proposed model consists of three detection types that can lead us to intrusion detection in a system: anomaly detection, misuse detection and specification–based detection.

4.4.2.1 Anomaly detection models

According to anomaly detection model, unexpected behaviours illustrate/give indication for intrusions. Evidently, there must be a measure that defines the expected user or process behaviour. The anomaly detection model, especially, analyses a group of system features and compares the behaviour of these features with a set of expected values for these features.

The concept of anomaly detection pertains directly to the concept of value deviation detection. This value deviation can deal with values (for system features) that don't agree or are out of the limits of a predefined set of reasonable values for the system. The value deviations are taken as anomalies. Labelling a value as anomaly implies that there is a method for labelling values as normal (accepted for the system under examination). This method is based on statistical models. Denning [81] described five different statistical models: Operational Model, Mean and Standard Deviation Model, Multivariate Model, Markov Process Model, Time Series Model.

4.4.2.2 Misuse detection models

In the area of the intrusion detection systems, the concept of the misuse refers to detection based on rules. Misuse detection, specifically, determines if a system command sequence violates the system security policy. In such case, a possible intrusion is described.

Misuse detection requires knowledge of all the vulnerabilities that occur in a system. This knowledge is concentrated in a rule set that constitutes a core and critical component of a misuse detection system. A misuse detection system applies these rules on data that have been provided to (or gathered by) the system, in order to decide if these sequences of data agree with rules of the set. If there is such an agreement, it is deduced that a possible attack is in progress. This category of intrusion detection systems is usually based on expert systems. Such a system can not detect attacks that are not known to rule sets designers. Recent detection systems use adaptative methods, like neural networks and Petri nets, to ameliorate the detection capabilities.

4.4.2.3 Specification – based detection

The specification – based detection method searches the space of a system states. When the system enters a state that is known to be unwelcome, a possible intrusion is reported. Specification – based detection defines if a command/process sequence violates or not the normal execution of a program or of a whole system.

For security reasons, only the programs that can change, in any way, the protection state of the system, should be allocated and checked. Specification-based detection is based on either traces, or event sequences [148] and is on its initial steps. Between its positive and innovative features, one can stand on the formalization of the events that could happen. By this means, the unknown attacks could be detected. However, this method requires great effort for the detection and the analysis of the programs that could raise security issues.

Ko et al. [148], developed a specification – based detection model for UNIX. They identified aspects of program behavior that are relevant to security: access of system objects, process sequencing, program synchronization and race condition (a special problem in program synchronization). Their model is based on traces and a formal notion of monitored subjects. They developed a prototype of a specification-based intrusion detection system that detects attacks exploiting the vulnerabilities of privileged programs in UNIX. An important aspect of their research is that they developed a language framework, parallel environment grammars, for specifying trace policies (capture the intended behavior of a program). The systematic methodology for developing trace policies for programs may also be useful in future research on developing overall security policies for computer systems and networks.

4.4.2.4 Intrusion Response

Once an intrusion has been detected, the next major research issue was how a system should response in this case. The main goal is: the attack to be faced with the minimum possible impact on system, as it has predefined by the security policy of the system. IDS generate alerts notifying administrators of this fact. The next (response) step is undertaken either by the administrators of the system or the IDS itself, by taking advantage of additional countermeasures (specific block functions to terminate sessions, backup systems, routing connections to a system trap etc.) – following the system security policy.

Ragsdale et al. [202] have made a research and a list of all existing intrusion detection and response systems. Most of intrusion detection and response systems are notification systems - systems that generate reports and alarms only. Some systems provide the additional capability for the system administrator to initiate a manual response from a limited pre-programmed set of responses. While this capability is more useful than notification only, there is still a time gap between when the intrusion is detected and when a response is initiated. Automatic response systems immediately respond to an intrusion through pre-programmed responses. With some exceptions, all of these automatic response systems use a simple decision table where a particular response is associated with a particular attack. If an attack occurs, a pre-programmed response executes. These pre-programmed responses consist predominantly of the execution of a single command or action instead of the invocation of a series of actions in order to limit the effectiveness of the adversary.

One of the exceptions of the automatic response systems is Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [201]. EMERALD resolver – an expert system - is the component that has the responsibility to receive reports for resource objects under attack from activities analysis EMERALD components and to revoke the various response handlers that have been defined for the object that reports refer to. Different responses are associated with different suspicion levels and the system adapts its responses based on the degree of suspicion.

4.4.3. Intrusion Prevention Systems

Firewalls, as mentioned above, can prevent attempts to access the internal network by blocking specific addresses while Intrusion Detection Systems can identify and alert the user for a possible attack as it occurs. Both technologies are critical for the defence mechanism of a system, but they both have limitations. A firewall cannot scan and evaluate the contents of every package in an efficient way in order to block suspicious packages. An IDS, in contrast, can scan and evaluate traffic that passes through but cannot do anything to stop it. Intrusion Prevention Systems (IPS) come in this point as proactive defence mechanisms that are able to detect malicious packages, through a comprehensive scan, and stop an intrusion before it harms the system. They basically combine the blocking capabilities of a firewall with the traffic inspection of an IDS.

Some use the IPS term to describe the next-generation of IDS systems that will be able to block certain kinds of attacks. Others use the same term more broadly and include firewalls, for instance, in the intrusion-prevention category, since firewalls can block certain attacks. IPSs are designed to sit in-line on the network and monitor the network traffic, just like IDSs, but when an event occurs can take an action and based on predefined rules.

There are two main categories of Intrusion Prevention Systems: Network-based IPS (NIPS) and Host-based IPS (HIPS):

Network-based IPS (NIPS)

A NIPS is an in-line device, between the Internet and the internal network, that can make decisions on whether or not to allow packages from the Internet to pass into the internal network based on attack detection. It actually combines features of a standard IDS (attack detection) and a firewall (blocking capabilities).

As an in-line device, a NIPS has at least two network interfaces, one for the internal network and one for the Internet. Packets reaching each interface are being examined whether or not they may pose a possible threat. The threat detection is based on methods of signature detection and anomaly detection. The content of each packet is examined for known signatures of threats or for unusual content. If a possible threat is detected, the NIPS in addition to alerting the user for the threat, it will automatically block the packet and mark the specific flow of packets as a threat. Thus, all the remaining packets from this flow will also be blocked. Packets that don’t pose a threat will be passed to the other network interface.

In order to prevent and stop unknown threats before their deployment an anomaly detection method is used. This method is based on the previous knowledge of the specific protocol or the usual behaviour of the specific application. Any packet that does not act according to this knowledge is treated as a threat. This system has the drawback of only being able to protect certain protocols and applications that are in wide use.

Host-based IPS (HIPS)

Host-based IPSs reside on servers and workstations. They are monitoring the hosts' application actions and calls to the system in order to detect a prohibited or unusual action. The methods they use are based on the signature detection of known viruses or malicious programs and anomalous or irregular behaviour of the system. In order for a HIPS to specify an abnormal behaviour, a policy that specifies the normal behaviour of the supporting operating system or application is provided. Any behaviour that doesn't align with this policy is treated as an irregular behaviour.

The attacks that host-based IPSs protect against include viruses, spam, spyware, worms, Trojan horse programs, key loggers, bots, buffer overflows and denial of service attacks.

Host-based IPS can also provide protection from internal attacks such as the installation or execution of a malicious program from a legitimate user.

Desai [82] introduced another classification, distinguishing among five types of IPSs:

— In-line NIDS: this type of IPS works exactly like the NIPS described above. It's an in-line device that monitors all the traffic between the internal and external network. It uses signature or anomaly detection methods in order to pick out possible threats while it works in a transparent way for both the legitimate users and intruders.

— Application-based firewalls/IDS: the combination of application firewalls and IDSs is usually marketed as an intrusion prevention solution. Both application firewall and IDS must be loaded on each server that is to be protected. This kind of IPS is customizable to each server and application that needs to be protected because there must be a profiling/training phase before the protection phase. During the profiling phase, the IPS can watch the user's interaction with the application and the application's interaction with the operating system to determine what legitimate interaction looks like. After the IPS has created a profile, a policy is constructed and it can be set to enforce it. This type of IPS offers one of the greatest amounts of protection for custom written applications because one can customize each policy so that it offers the greatest amount of protection since each application firewall/IDS is loaded on each physical server you.

— Layer seven switches: Layer seven switches are devices that work as the usual network switches but in their case they work in the application layer. They are usually used to balance the load of an application across multiple servers by inspecting the protocol of every packet and forwarding the packets to specific servers according to predefined rules. The intrusion prevention capabilities they offer are limited in denial of service attacks detection and prevention. They work using a signature detection method for stopping these attacks and without affecting the network performance, thus guaranteeing speed and uptime.

— Hybrid Switches: This type of IPS is a combination of the host-based application firewall/IDS and the layer seven switches. These systems are implemented in hardware, located in front of the servers, like the layer seven switches, but instead of using a set of predefined rules, they use a policy similar to the application firewall/IDS. They inspect packets for malicious content defined by the policy.

Deceptive Applications: This type of IPS first observes all the network traffic and figures out what legitimate traffic looks like, similar to the profiling phase of the application firewall/IDS. Then, when it sees attempts to connect to services that do not exist, it sends back a marked response to the attacker with some bogus data so that when the attacker tries to exploit the server the IPS will see the marked data and stop all traffic coming from the attacker.

4.4.4. *Access Control Models*

Access control models provide high-level, domain-independent, and implementation independent reference models for the architecture and design of access mechanisms.

Historically, access control models are classified in two broad categories: mandatory [189] and discretionary [189, 190]. Later on the need for a more flexible access control model, which it would be suitable for big organisations, has lead to the role based access control model and the context based access control model. We will describe each model in the next section and highlight their characteristics.

4.4.4.1 Mandatory Access Control

Mandatory access control governs the access of objects by subjects by using a classification hierarchy of labels. Every subject and object is assigned a label. All access is based on comparisons of these labels and, in general, is statically enforced. We say that access control is mandatory because the system centrally enforces all decisions to permit a subject's activities based on labels alone. Entities have no say in the matter.

Mandatory access control centralizes the knowledge base used to make decisions, although subjects and objects can negotiate access based on local information. Entities are allowed to read objects with lower classifications and can write to objects only with the same classification level.

4.4.4.2 Discretionary Access Control

Discretionary access model organizes the security of a system into a two-dimensional matrix of authorizations in which each subject-object pair corresponds to a set of allowed access modes. The access modes in the matrix can be modified through commands.

Discretionary access control governs the access of objects by subjects based on ownership or delegation credentials provided by the subject. These models are implicitly dynamic in that they allow users to grant and revoke privileges to other users or entities.

Once access is granted, it can be transitively passed onto other entities either with or without the knowledge of the owner or originator of the permissions. Discretionary access control models enable subjects to transfer access rights for the objects they own or inherit, or for which they have received "grantor" privileges.

Discretionary access control is flexible, but the propagation of rights through the system can be complex to track and can create paradoxes.

4.4.4.3 Role Based Access Control

Role based access control (RBAC) has its roots in historical practices that predate its formal model, except that RBAC's features stem primarily from the commercial world. Also like multilevel security, RBAC is conceptually simple: Access to computer system objects is based on a user's role in an organization. Roles with different privileges and responsibilities have long been recognized in business organizations, and commercial computer applications. From the late 1980s until now several access control models [43, 86, 188, 235] have been introduced based on the conception of the role. The two major proposals from [97, 211] prompted NIST to initiate an effort to establish an international consensus standard for RBAC which it was finally published in the ACM RBAC workshop in 2000 [212].

The proposed standard is divided in two sections: the reference model and the functional specifications. The reference model section defines the sets of the basic RBAC elements (i.e., users, roles, permissions, operations, and objects), the relations between them and the functions that are included in this standard. Additionally it provides a separation between all the available features of RBAC defining four packages. The Core RBAC package with the basic RBAC features, the Hierarchical RBAC package for supporting role hierarchies, the Static Separation of Duty package for avoiding conflict of interests by enforcing constrains on the assignments of the roles and the Dynamic Separation of Duty Package which can avoid conflicts of interest by imposing constraints on the roles that can be activated within or across a user's session. The functional specifications section defines the functional requirements for every package of the RBAC model (i.e. functional requirements for administrative operations and queries for the creation, maintenance, and review of RBAC sets and relations)

The pervasiveness of RBAC's application within modern day IT infrastructures is significant. Today, RBAC features are included at all levels of enterprise computing, including operating system, database management system, network, and enterprise management levels. RBAC is being incorporated and integrated within infrastructure technologies such as public key infrastructure (PKI), directory and Web services.

4.4.4.4 Context Based Access Control

Context based access control (CBAC) models leverages and extends the power of traditional role based access control models by taking access control decisions based on the combination of the required credentials of users and the context and state of the system. RBAC systems required only user credentials in order to assign a role to the user and then permit or deny the access to an object. CBAC requires, additionally to those credentials, dynamic information such as the physical location of the user, date and time of the access control request, the state in which the desired object is, etc. Those variables consists the context information, which the decision system requires to take in account before it delivers a decision.

Several proposes and implementations have been introduced the last years based in the conception of the context based access control, but none of them have been recognised as a standard yet. Covington, Moyer and Ahamad [73] are from the first who tried to expand the RBAC model by introducing roles for subjects, objects and the environment of the system. These three types of roles consists the context information needed for the access control decisions. Also Kumar, Kamik and Chafle [150] later expanded the RBAC model by introducing the notions of role context and context filters. Context filters are Boolean expressions based on the context information of the user attempting to get authorized, as well as the context information of the object upon which this authorization is attempted.

Srinivasan et al. [226] proposal was focused on context-aware applications and how they can take access control decisions in a ubiquitous computing environment while Corradi, Montanari, Tibaldi

[70] presented a context-centric access control middleware, called UbiCOSM, that is suitable for implementation of services in such an environment.

Another approach from [125] presented a way to use context information (location specific) to provide anonymous access to services without limiting the ability of the service provider to impose various security levels. They provided a list of methods that can verify the user's claimed authenticity in various ways and degrees.

Other proposals for using CBAC in specific environments delivered by Wilikens et al. [249] who described the requirements for authorization and access control within a healthcare environment based on an extended model of RBAC which incorporates contextual information and can be integrated in a wider trust infrastructure including the use of Smart Cards and Cholewka, Botha and Eloff [61] who proposed and implement a context-sensitive access control mechanism within a workflow environment.

4.4.5. Conclusions

We presented the current approaches of security and cryptographic mechanisms used by the majority of the information systems. All those mechanisms provide ways to protect and maintain the three basic principles of security, confidentiality, integrity and availability. However, in many cases these mechanisms are not enough to guarantee those principles.

Many security protocols, using cryptographic techniques, have been proposed in the literature to achieve several goals, e.g., authentication and secrecy. They are supposed to succeed even in the presence of malicious entities that can interfere with their correct execution. Unfortunately, many of them may fail when an intruder intercepts some messages and exploits the information they contain. The failure of such protocols sometimes lies in the implementation environment of the protocol that is not verified for full correctness.

Firewalls have also many weaknesses. Packet filtering firewalls suffer from a number of weaknesses, as described by Chapman [58]. Among them is the complexity in the specification and verification of the rules; the shortage of the audit capabilities; and the possibility to get around the filtering policy (e.g. a site system's telnet server that normally listens at port 23 could be told to listen at port 9876 instead; users on the Internet could then telnet to this server even if the packet filter blocks destination port 23).

Weak points can also appear in Application level gateways. The main problem is their slow speed. According to their design, every packet has to pass through them and the contents of every completed connection are examined in detail. Such process adds, of course, more time to complete. Moreover, they cannot protect against internal threats and they are also unable to protect against the transfer of unknown malicious programs, which they do not look harmful during their admission but they actually contain hostile code within.

Intrusion detection systems (for example, those surveyed [22, 29, 224] suffer from at least two of the following problems: First, the information used by the intrusion detection system is obtained from audit trails or from packets on a network. Audit data has to traverse a longer path from its origin in order to reach the intrusion detection system and be analysed. During this process, it can potentially be destroyed or modified by an attacker. Thus, the intrusion detection system, which has to infer of the behaviour of the system from the data collected, can conclude to real time system's behaviour misinterpretations.

Second, due to the satisfaction of the desired intrusion detection system property, i.e. to run continuously, an IDS has to utilize system resources all the time even when there are no intrusions occurring.

Third, because the components of the intrusion detection system are implemented as separate programs, they are susceptible to tampering. An intruder can potentially disable or modify the programs running on a system, rendering the intrusion detection system useless or unreliable.

Finally, although access control mechanisms seems to be useful for checking static information, their evolution has shown that more dynamic properties of the entity which requests authentication are needed to make sure access control decisions.

4.5. Next Generation Intrusion Detection Systems

4.5.1. Distributed Intrusion Detection System (DIDS)

DIDS [222] combines the capabilities of an intrusion detection system designed for a network with a monitoring capability for intrusion in distributed systems. A network intrusion system has been introduced in [120] under the name of Network Security Monitor (NSM).

The NSM system has been developed to run in LANs. In the beginning, it creates a profile of the network regarding the expected network utilization and then, in the operation, it compares the network utilization with the one of the profile. In addition, the NSM allows a set of valid signatures to be predefined, in order to detect suspicious network utilization that may lead to attacks. The NSM can be configured to monitor a user, a group of users or a service activity and to record potential anomaly.

The NSM monitors the source of network utilization, the destination and the provided service. It defines a unique connection ID for every connection. Sources, destinations and provided services constitute the base of a three dimensional table. Each element of this table illustrates the number and the overall payload of packets that were sent through this connection during a predefined time period. In addition, the NSM estimates the expected overall payload of this connection. Table data and the NSM estimations are compared and any table element that is out of the estimated value range is interpreted as an anomaly.

NSM developers discovered that a great amount of data was produced during a network analysis. For reducing the respective cost, they established a hierarchy for the table data and respectively for the estimated expected values. If there is an anomaly in any data set of the hierarchy, the security administrator of the system can configure the NSM to analyze the set (in which there is an anomaly) in lower levels. A hierarchical data set, for instance, could be constituted by transferred data through two systems connection for every service, e.g. {(A, B, SMTP), (A, B, FTP)…} where A, B are the connected systems. In the higher level, the NSM would analyze the source data. If an anomaly occurred, the NSM could investigate and define in which exact source – destination pair the anomaly occurred.

Based on the NSM data table, a simple signature – based scheme for misuse pattern identification was developed. For instance, repeated telnet connections which last as long as the standard access time could give an indication for failing login attempts. A definite rule could search for the above incident among the table elements. Heberlein et al. [120] developed a number of rules for network utilization. Among those were rules for dealing with monitoring for excessive amount of login attempts and for the connections of one system with fifteen or more other systems.

The NSM is important for two reasons. A lot of posterior IDSs were based on it. In addition, network intrusion detection is feasible in a practical level, as it was illustrated the NSM implementation. While network traffic is distinguished by encrypted messages exchange/flow to such a degree that analysing capability of packet content fades, NSM analyses the traffic itself and not the packet content.

With the evolution of possible attacks on systems, it was concluded that monitoring of network utilization alone and, on the other hand, the monitoring of a single host alone were insufficient. An intruder who tries to connect to a system through a login account that does not require password, is not detected as malicious by an IDS dedicated to network monitoring (e.g. NSM). Once the adversary has access to the system, he might exhibit behaviour that would have alerted most of the existing single host IDSs (e.g., changing passwords and failed events). Similarly, if an intruder tries a few common account and password combinations on each of a number of LAN computers, the IDS on each host may not flag the attack. On the other hand, an IDS dedicated to network utilization monitoring could detect the repeated failed logins.

The DIDS architecture, combines distributed monitoring and data reduction with centralized data analysis. The DIDS components are the DIDS director, a single host monitor per host and a single LAN monitor for each broadcast LAN segment in the monitored network.

The DIDS director consists of a communication manager (responsible for data transfer between the director and the monitors), a rule - based expert system (that evaluates and reports the security state of the monitored system by making inferences based on the events that monitors send) and a system security officer (that pertains the critical management component).

The host monitor consists of a host event generator and a host agent. In the DIDS prototype that has been implemented on Sun SPARCstations running SunOS 4.0.x, host monitors take advantage of the C2 security package functionality (through this package, the OS produces audit records for virtually every transaction on the system). Based on these audit records, host event generators create events that are sent to the expert system for processing. The format of an event is: significant data provided by the audit record, plus action (e.g. session_start or end, read, write, etc.), plus object (e.g. authentication, network, sys_info, etc.).

The LAN monitor's responsibility is to observe all of the traffic on its segment. In order to monitor host-to-host connections, utilised services and volume of traffic can be especially helpful. The LAN monitor reports, which are sent to the DIDS director, have to do with network activity (e.g. rlogin connections), security-related services utilization and deviation from network traffic patterns.

Snapp et al. [222], gave a solution to one of the more interesting issues for intrusion detection in a networked environment; the tracking of users and objects (e.g. files). They have addressed the multiple user identities problem by creating a network-user identification the first time a user enters the monitored environment.

Another point that is of significant importance is the utilization of a rule based expert system with learning capability in the DIDS prototype. The expert system applies the rules to the evidence provided by monitors. Each rule has a rule value that can be changed after configuration. This value illustrates the confidence that the rule is useful in detecting intrusions. If a report for an intrusion made by the expert system was faulty, the expert system lowers the rule values of the associated rules that were used to draw that conclusion.

4.5.2. Autonomous Agents for Intrusion Detection (AAFID) System

By investigating intrusion detection approaches from fault tolerance perspective, Crosbie and Spafford [75] drew the conclusion that the monolithic IDS approach represents a single point of failure. By attacking successfully the IDS, protected system security is greatly reduced. Thus, they proposed that it would be useful to distribute the IDS functionality across multiple independent (and simple) components, autonomous agents, in a networked environment.

According author's approach, the internal design of the agents is based on the genetic programming paradigm (that is a powerful machine learning paradigm and allows both feedback learning and discovery to agents).Each one of the autonomous agents that are used is responsible for monitoring one small aspect of overall system. When an agent detects any misuse, anomaly or specification deviation, it reports the evidence to the other agents. All of the agents that operate on the system and cooperate together, then, can define if the reported evidences were enough to indicate a possible intrusion.

The most important aspect of this architecture is the cooperation of the independent agents. Thus, there is not, any longer, a single point of failure. If an agent stops running, the rest of them can carry on. Moreover, if an adversary takes an agent under control, no critical information about the other agents is revealed to the adversary.

AAFID system [23] implements the above approach. Each host of a network can contain any number of agents that monitor for interesting events occurring in the host. All the agents in a host report their findings to a single transceiver. Transceivers are per-host entities that supervise the operation of all the agents running in their host. The transceivers report their results to one or more monitors. Each monitor oversees the operation of several transceivers. Monitors have access to network-wide data, therefore they are able to perform higher-level correlation and detect intrusions that involve several hosts. Monitors can be organized in a hierarchical fashion such that a monitor may in turn report to a higher-level monitor. Also, a transceiver may report to more than one monitor to provide redundancy and resistance to the failure of one of the monitors. Ultimately, a monitor is responsible for providing information and getting control commands from a user interface.

The main advantage of AAFID architecture is that can be easily extended configured and modified. On the other hand, it faces many problems that always occur in the area of distributed systems, such as performance (especially from a host perspective) and security (referring to means for securing the communication among the independent components of the distributed IDS).

4.5.3. Immunology-based Security

The Artificial Immune Systems (AIS) are one of the biologically inspired computing paradigms like Neural Networks and Genetic Algorithms. The aim of biologically inspired computing is to design systems to solve complex problems taking inspiration from mechanisms (problem solving techniques) evolved over thousands of years by biological systems to cope with the complexity of natural world.

An overview of basic theory of Artificial Immune Systems and application based on this paradigm can be found in [269].

There are various features that make natural immune system an appealing source of inspiration for designing computer security systems [275]:

- Multilayered protection: a breaches or attack occurring at one level can be detected by the next layer;
- Distributed detection: the detection and memory system is highly distributed and without a centralized control;
- Unique copies of detection systems (diversity): each individual has a unique set of characteristic; there is always the chance that some individual in the population is able to resist to an attack due its specific characteristic;
- Detection of previously unseen foreign material: the natural immune system is able to recognize new forms of infections;
- Imperfect detection: the natural immune system is able to detect foreign entity even if it doesn't posses a detector perfectly matching that entity;

In the context of Computer Security the AIS paradigm has been investigated in three main area of application:

- anomaly detection;
- virus detection and elimination;
- network intrusion detection. (Protection from network-based intrusions)

In the following section we review the most relevant works in each area.

4.5.3.1 Virus detection and elimination

One of the key design principle adopted in Artificial Immune Systems is the principle of the Self/Non-Self discrimination. According to this principle the Immune System is interpreted as a system able to distinguish internal components (*self*) from external entities (*non-self*) and, therefore, the problem of protecting computer systems can be reformulated as the problem of learning to distinguish self (legitimate users, authorized actions original source code etc.) from other (intruders, computer viruses, Trojan horses, etc.).

One the first application area of application of the principle of Self/Non-Self discrimination has been the computer virus detection and elimination [271, 281].

The main goal of the work described in [281] was to create a system able to automatically create and promptly distribute anti-virus data to networked hosts. The main motivation was the need to contrast the increasing speed of diffusion of computer viruses enabled by open networking environments by removing humans from the response loop to computer virus infections.

The system was designed after the two main components of Biological Immune Systems:

- *Innate immune system*: a first line defence having knowledge of both the self and of some broad classes of harmful entities. This component provide a generic defence to ongoing attacks;
- *Adaptive immune system*: a learning system able to produce a strong response very specific to the ongoing attack; this component is activate by the activation of the Innate Immune System.

The proposed approach consists of the following steps:

4. Discovering of previously unknown virus on a infected hosts;
5. Capturing a sample of the virus and sending it to a central computer;
6. Analysing the virus automatically to derive a prescription for detecting and removing it from the infected hosts;
7. Delivering the prescription to the monitored hosts and running the local copy of the anti-virus to detect and remove all occurrences of the virus;
8. Disseminating the prescription to all other computer on the network.

In the above procedure steps 1 and 2 pattern the innate immune system whilst 3,4 and 5 the adaptive immune system.

In step 1 the new unknown viruses are discovered exploiting two alternative approaches: Generic Disinfections heuristic and a classification system.

The Generic Disinfections heuristic is an implementation of Self/Non-Self discrimination principle:

1. When a new program is installed a fingerprint is computed and stored in a database. This fingerprint represent a compact description of the "self".
2. During the monitoring the fingerprint of the program is recomputed and compared with the one (the "self") stored in the database. If a mismatch arise a contamination is detected i.e. a "non-self" is in the system.

The Biological Immune System is a source of inspiration also in step 3. The derivation of a prescription is done using decoy programs. The virus discovered in step 1 is executed in a controlled environment in order to make it infect a diverse suite of "decoy" programs. The infected "decoy" programs are the basis to extract the virus signature by means of pattern matching. Digestion and exposition of its foreign proteins (antigens) is indeed one of the core strategies of the Adaptive Immune System.

Indeed one of the first relevant works to suggest the use of the Self/Non-Self discrimination principle to computer security is [271]. The work introduces a change detection algorithm inspired by Immune System and applies it to the problem of detecting computer viruses.

The proposed algorithm (negative-selection algorithm) runs in the following stages:

1. define *self* *S* as a collection of binary strings of length *l* over a finite alphabet representing *D* the data to protect;
2. generate a set *R* of detectors each of which fails to match any string in *self*;
3. monitor *self* for changes by continually matching the detectors *R* against S. If some string in R matches S then S has been corrupted.

The key point in the above algorithm is that in the monitoring step (3) the matching is done against a description of the non-self (derived in step 2) hence the name of the algorithm. The step 2 is inspired by the censoring activity taking place in the thymus where, to prevent autoimmunity, T cells reacting with the normal occurring patterns in the body are destroyed.

Another key point is that the matching is a partial matching:

Although counterintuitive the important mathematical result is that even with a small number of detectors a random change in S has a very high probability to be detected. Moreover the size of R can remain small as the size of S grows. In recent years a formal framework has been proposed to analyse the tradeoffs between positive and negative detection schemes [270].

The claimed advantages for the negative-selection algorithm are:

- The checking activity can be distributed;
- The quality of checking: the probability of a change detection can be traded off against the cost of checking;
- Protection is symmetric S and R protect each other: S can be used to monitor changes in R;
- Due to the computational complexity of generating detectors it is hard to alter detectors to hide change in S;

4.5.3.2 Anomaly detection

In a later paper [272] propose to use a similar approach based on positive selection to address the problem of anomaly detection in Unix systems.

In order to apply the immune-system inspired approach the first issue to be addressed is how to define a suitable "self" in a computer system. The solution is complex since:

- System configuration changes;
- Users change their personal work habits;
- New users and new machines are added to computer networks.

The paper adopts the normal behaviour of a Unix process as "self": short sequences of system calls represent a stable signature for normal behaviour. The experiments conducted by the authors show that such signatures:

- Have low variance over a wide range of normal operating conditions;
- Are specific to each different kind of processes;
- Have high probability to be perturbed when attack or attacks attempts occur.

The proposed algorithm runs in to phases:

1. Training phase: for each process a database of normal patterns are created by scanning traces of normal runs; [288] compares alternative schemes for representing normal patterns: stide, t-stide, RIPPER, HMM.
2. Monitoring phase: traces are scanned looking for sequences that are not present in the normal database.

The expected advantage from the approach is that having a simple definition of normal behaviour (*self*) enables on-line monitoring that runs in real-time (the operating system could perform the check at each system call without affecting performances). Moreover since the normal behaviour is created by tracing normal runs of the program the approach doesn't require determining a behavioural specification from the program code. The preliminary work consider only privileged processes (`sendmail`, `lpr`) since:

- they are likely more dangerous than other programs (they can access more parts of the computer system);
- they have a limited range of behaviours;

- they behaviour is stable over time.

The recognized disadvantages of the approach are:

- It ignores the parameters of system calls;
- It ignores timing information (unable to detect attacks relying on race conditions);
- It ignores instruction sequences between system calls.

4.5.3.2.1 Beyond Self/Non-self discrimination

The work of [277] proposes a real-valued negative selection (RNS) algorithm to overcome some of the drawbacks in the original binary version [276]:

- Scalability: a large number of detectors has to be generated as grows the required detection accuracy;
- It is difficult to map detectors to domain space: hence it is difficult to explain where is the problem once it has been detected;
- There is a sharp distinction between self and non-self;
- Binary representation of self makes difficult to integrate the negative selection algorithm with other algorithms.

The main distinctive features of the RNS algorithm are:

- The detectors are a couple defined by a n-dimensional real vector and real value hence detectors are hyperspheres in R^n;
- The matching rule is defined by a fuzzy membership function depending on the Euclidean distance from the detector and the radius of the detector.

The RNS algorithm is used in [277] as basis to define an hybrid approach to immunology-based anomaly detection: the RNS algorithm is used to generate abnormal samples from normal samples; both normal and abnormal samples are then used to create a classifier by means of a supervised algorithm.

By adopting a fuzzy membership function the RNS algorithm represent one of the first tentative to depart form the Self/Non-Self principle in the context of the Artificial Immune System. This departure indeed reflects also the evolutions in the Immunology area. These evolutions are motivated by the fact that the Self/Non-Self model is not able to explain the observed behaviour of the Biological Immune System when the biological “self” changes [323].

Indeed it can be said that the RNS algorithm could be seen as the digital counterpart of the Infectious-Nonself principle. The Infectious-Nonself principle postulates the existence of different classes of Non-Self and that the Immune System activates an immunitary response only against some of these classes (Infectious-Nonself).

One of the most controversial evolutions in the interpretation of the Immune System is the Danger Theory. According to this theory an immunitary response only when detection of non-self co-occur with the detection of danger signals produced by cells under stress conditions (as an injury).

In recent years the Danger Theory has been proposed to overcome some of the limitations of the Self/Non-Self model widely adopted in the Artificial Immune System [259].

The Danger Theory is a source of inspiration in [282] to design a system able to extend a policy for the systrace system call policy checker. In the system the danger signals controlling the activation of an appropriate response are:

- Security policy violation;
- Highly fluctuating process cpu or memory usage;
- System load average.

4.5.3.3 Network intrusion detection

The Artificial Immune System paradigm has received attention also for the development of Intrusion Detection Systems [259].

The LISYS system described in [278] address the problem of detecting anomalous network traffic in a broadcast Local Area Network (LAN). LISYS is an implementation of the conceptual framework ARTIS inspired by Immune Systems and described in the same paper. The main components of the ARTIS framework are:

- Detectors: to represent a subset of the non-self;
- Anomaly detection to detect non-self string that have not been encountered previously;
- Sensitivity levels to increase the ability to detect divers non-self strings;
- Memory detectors: to improve detection of previously encountered non-self;
- Activation thresholds: to reduce false positive;
- Co stimulation to reduce false positives by eliminating auto-reactive detectors (detectors matching elements belonging to self);
- Tolerization: to reduce false positives;
- Multiple representation to improve detection rates when the relevant non-self strings are similar to self strings;
- Finite lifetimes: to ensure that gaps in detection coverage are not static and predictable;

Within LISYS the self of the network is represented by a collection of “normally occurring” connection triples (“datapath”) having the form (sender IP address, receiver IP address, receiver TCP port) obtained monitoring TCP SYN packets. Normal occurrence of a connection triple is defined by observing the network traffic over a certain amount of time.

The architecture of the system is distributed each node in the network runs a detection node. Since the network is a broadcast network each detection node sees every packet.

Each detection node contains a different set of detectors each consisting of a binary string (binary representation of a datapath). Each detector can assume one of three states: immature, naïve or memory.

Detection of anomalous packet is performed trying to match packets against a set of detectors of a detection node. Packets are matched against detectors using r-contiguous bits matching rule: a detector matches a packet if they have r contiguous bits in common. Each time a packet is matched against a detector the state of the detector changes according to the transitions of a Finite State

Machine. Matching that move the matched detector to the activated state indicate that the packet belong to non-self (negative selection).

The system runs in two phases:

1. Training phase: during this phase an initial set of detectors is created at each node by exposing the system to events generated by normal traffic (the system is not under attack). During this phase all detectors matching any packet are discarded (negative selection) and replaced by a new one randomly created. At the end of this phase each detection node receives a set of naïve detectors;
2. Monitoring phase: during this phase each detection node performs the following operations:
    a. Tries to match received packets against its set of detectors. If the packet doesn't match any detector the packet is considered to belong to the self and hence it is ignored. If one or more matches occur the state of the matching detectors is changed according to the lifecycle defined in ARTIS. If one or more matching detectors reach the activated state a human operator is asked to decide if the packet belongs to non-self (i.e. if it is an anomaly resulting from an attack) or it represent a false positive. Detectors that generated false positive are discarded.
    b. Discards naïve detectors did not exceeded activation threshold during lifetime.
    c. Generates new randomly created detectors and tries to match against the self.

One of the major criticisms to the negative-selection algorithm is the high computational complexity required to generate the set of detectors [322]. Indeed one of the most active research areas for the negative-selection algorithm is the quest of scalable algorithms to generate detectors. However it is worth noting that the scalability problem reported in [322] strongly depends on the choices made for the alphabet (10 letters) and the r parameter of the matching rule [260].

The AIS paradigm is used in [283] to detect node misbehaviour in ad-hoc network running the Dynamic Source Routing protocol (DSR). The work proposes a mapping between the components of AIS and the elements of an ad-hoc network. The proposed system monitors sequences of protocol events collected over limited period of time and having a maximum length. In order to avoid performance issues the sequences of events are encoded using a scheme counting the occurrences of predefined regular patterns within the sequence.

When a non-self sequence of event is detected the node is classified as “detected”. Eventually a node becomes “misbehaving” if in average the number of time the node is classified as “detected” is more of a given threshold.

In the area of Network Intrusion Detection the Danger Theory has been proposed [282] to design a system for automated Worm Detection and Responses. The key requirement for the described system is to keep low the number of false positive to avoid that unnecessary automated counter measures compromise the functioning of the system. More in general the requirement is to keep the response adequate to the attack severity and certainty.

4.5.3.4 The Immune System as a Multi-Agent System.

The natural immune system is also been taken as a model to organize Multi-Agent System for intrusion detection and response. The main reason residing on the similarities that can be drawn between the two models:

- They are distributed systems with decentralized control;
- Their components are (semi) autonomous entities;
- They have capabilities to learn from their experience;
- Their components coordinate and communicate;
- They are able to adapt to changes in the environment;

The SANTA system (Security Agents for Networks Traffic Analysis) [267] is a Multi-Agent System that organize the society of agents according to concepts burrowed from the Immune System:

- *Monitoring agents*: these agents monitor computational nodes looking for specific anomalies (e.g. unusual user behaviour patterns, unusual usage of computational resources, invalid processes and priority violations;
- *Communicator agents*: they carry messages to other agents in the system;
- *Decision/Action agents*: they take decisions and perform task in order to enforce security policy. Tasks may activate one of the following type of response agents:
    - Helper agents: they report the status of the environment to the end user;
    - Killer agents: they are in charge to perform action like node shut down, kill processes, discard streams of or disconnect user sessions;
    - Suppressor agents: they role is mainly to suppress the action of other response agents in order to prevent actions due to false positive detections.

The MMDS system (Multi-level Monitoring and Detection System, [268]) follow a similar organization targeted for anomaly detection in ad-hoc networks. In MMDS Decision agents take their decision using fuzzy rules and the rule are evolved using genetic algorithms.

The CDIS system [326] is another MAS model of adopting an organization inspired by the Immune System. The main distinctive features of CDIS are:

- The distinctive features that define the self are 28 instead of the 3 of LISYS;
- All TCP, UDP and ICMP packets are monitored.

The AISIMAM system [325] uses the Jerne's model of immune network to organize a society of agents. Self-agents and Non-self agents share a common environment where they operate.

4.6. Open research issues for dynamic verification

Based on our survey, the open research issues related to dynamic verification relate to:

- The need to provide support for transforming of specifications of security and dependability properties that need to be monitored at runtime into the event patterns that should be observed to verify them.
- The need to develop mechanisms that can support the diagnosis of the reasons underpinning run-time violations of security and dependability properties requirements that could inform system adaptation to ensure that violations will not re-occur.

- The ability to support the specification of end-user personal and ephemeral security and dependability properties, the automatic assessment of whether or not such properties can be monitored at run-time, and the transformation of these properties onto monitorable patterns of run-time events.
- The development of techniques that would allow the identification of scenarios of potential security and dependability threats (i.e. potential violations of security and dependability properties that have not occurred yet but seem to present a realistic possibility for the subsequent operations of a system) and the translation of these scenarios into monitorable event patterns that would allow the development of pro-active techniques for protecting security.
- The need to develop mechanisms that can ensure that the events used in dynamic verification have not been altered by an attacker in order to affect the results of the verification process and consequently the recovery actions that may be taken in response to these results.

An additional, but by no means less important, open issue in dynamic verification is the need to develop monitors that could detect violations of security and dependability properties efficiently and in time that will allow the effective reaction to such violations.

5. Recovery

Research in security and dependability has been primarily concerned with the prevention and detection of threats and system faults rather than recovery from them when they occur. The ability, however, to recover from security attacks or system faults identified at run-time and possibly adapt in order to handle these attacks or faults is also increasingly recognised as an important requirement related to the security and dependability of systems. Recovery capabilities are required to ensure that the scale of an attack is minimised, the system's critical functionality is preserved and a system is guarded from further attacks. Such capabilities have been studied extensively in the areas of safety-critical systems (fault-tolerance) [14] and databases [36].

In this section, we provide an overview of recovery approaches and mechanisms that have been adopted and developed as part of run-time verification systems, safety-critical systems (concerned mainly with preserving dependability properties) and mission-critical distributed systems (concerned mainly with preserving survivability properties), and trusted recovery approaches developed as part of information warfare defence.

5.1. Recovery for run-time verification systems

Recovery mechanisms for modern programming languages are almost non-existent [168]. Usually, this task is left to the programmer that must implement them manually. Moreover, it is generally accepted that programmers are very bad at planning for fault recovery [248]. With this in mind, Manson et al. [168] have proposed a new programming language based approach, called RESCUE, for determining and recovering from faults, for applications developed using Aspect Oriented Programming (AOP). It is an extension of ApectJ [141] and aims to expose underlying features of the virtual machine in order to express faults in a variety of run-time constraints. RESCUE provides an asynchronous construct (the *mutex qualifier*) that can be combined with the code at runtime to handle exceptional conditions. The mutex qualifier is used on a plan to prevent that plan from executing concurrently with its parameter. A plan is a specification of fault conditions as a predicate over meters. Meters provide an interface to monitoring events.

Moreover, Manson et al. [168] present two ways of assisting the programmer with recovery:

— Checkpointing: This mechanism copies the program state periodically. They show a way of checkpointing relevant data selectively with aspects, in the customisable virtual machine. However, this approach is not accessible as it requires in depth knowledge of how the virtual machine works.

— Transactions: The aim of RESCUE is to support transactions in the following way. If a write transaction occurs, the original value of the heap location is written to a log. Thus, when a fault occurs, the original values can be restored from the log. Therefore, program flow is resumed from the beginning of the transaction or continued as if the fault never happened. RESCUE uses transactions in conjunction with programming language support for plans and meters.

d'Amorim and Havelund [78] have suggested that the integration of a system like Eagle and AspectJ could support temporal cutpoints where temporal Eagle formulae can function as triggers for actions to be executed. This would then be useful for developing fault tolerant programs that can change their behaviour when the temporal properties have been violated.

Kazman et al. [135] present an architecture model that allows a system to reason about its behaviour at runtime, by being self-reflective, and taking action in cases where it is required to change its behaviour, accordingly. This architectural model uses both runtime monitoring and abstraction, as well as codified knowledge of architectural styles, to develop a dynamic view of its architecture as it runs. *DiscoTech* is a system that is built to recover the architecture of systems, i.e. interpret the run-time behaviour in terms of architecturally meaningful events. Any analysis and repair of the run-time system is performed on the architectural model and not directly on the running system. No explanation of how this recovery is performed is given.

Feather et al. [97] present an architecture and a development process for monitoring system requirements (expressed as goals in KAOS) at run-time in order to reconcile the requirements with the system's run-time behaviour. The recovery phase is closely related to how requirement violations are managed and two such approaches are discussed [238, 240]:

1. At specification time, the developer anticipates as many obstacles to requirements as possible. Once these have been identified, the more robust specifications are defined taking into consideration the obstacles.
2. At run-time, violations of requirements are detected and resolved by making on-the-fly, acceptable changes to the requirements. By acceptable, they mean that the changes must satisfy the high-level goals underpinning the requirements that were violated.

Feather et al. [97] focus on the dynamic approach. The recovery step is the last step that happens at run-time and consists of analysing the violation file and applying the reconciliation tactic that was defined in the development level (statically). The reconciliation tactics are choices that have to be made for each breakable assertion between enforcing it and finding an alternative. Assertions may be constraints (an implementable goal), or assumptions (a fact taken for granted about agents in the environment of the system).

5.2. Recovery in safety-critical systems

Dependability of a computer system is defined by Avizienis et al. [21] as “the ability to deliver service that can justifiably be trusted”. The service that is delivered is the system behaviour as viewed by its users. A user is another system or human that interacts with the system via the service interface. A correct service is one that implements the system function. A system failure is an event that happens once the service deviates from correct service. Hence, a failure causes a transition from correct service to incorrect service. An outage is the delivery of an incorrect system. Another definition of dependability that they give in terms of failure is: “the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)”.

Dependability is achieved by addressing the following concepts: attributes, means and threats [21], as given in Table 5.1.

Attributes	Means	Threats
Availability	Fault prevention	Faults
Reliability	Fault Tolerance	Errors
Safety	Fault Removal	Failures
Confidentiality	Fault Forecasting	
Integrity		
Maintainability		

Table 5.1 – Dependability concepts [21]

Avizienis et al. [21] have also suggested that failures can be characterised further according to their domain, perception by system users and consequences on the environment of the system. These are given in Table 5.2.

Domain	Perception by two or more users	Consequences on environment
Value failures	Consistent failures	Minor failures
Timing failures	Inconsistent failures	Varying degree of failures …
		Catastrophic failures

Table 5.2 – The failure modes [21]

Faults can be classified according to six different criteria as given in Table 5.3. Malicious faults that compromise security fall under the intent category. These can be further divided into the following classes: malicious logics and intrusions. Malicious logics [312] include the developmental faults such as Trojan horse, timing bombs and trapdoors, as well as operational faults such as viruses or worms.

Phase of creation or occurrence	System boundaries	Domain	Phenomenological cause	Intent	Persistence
Development faults	Internal faults	Hardware faults	Natural faults	Non-malicious (accidental) deliberate faults	Permanent faults
Operational faults	External faults	Software faults	Human-made faults	Deliberate malicious faults	Transient faults

Table 5.3 – Categories of faults [21]

Techniques that have been used to combat faults and achieve dependability of a system under construction can be grouped as follows [128]:

— **Fault avoidance:** During requirements analysis, specification and design and later during maintenance (such as bug fixing, performance improvement), the methodologies used for system development should aim to produce a system that works as expected without the inclusion of errors (only a minimum of errors exist) [313]. Examples of such techniques include structured programming, information hiding, modularisation, etc. Also, firewalls and similar defences can be used for preventing malicious faults.

— **Fault elimination (detection and removal):** During requirements analysis, specification and design, and also during maintenance, faults in the system must be detected and removed by using techniques extensively, such as verification and validation techniques.

— **Fault tolerance:** Fault tolerance may be used during system execution in order to cope with run-time failures. In order for fault tolerance to be used, it has to be build into the system in the proceeding phases. A system is called fault tolerant with respect to a set of faults if it is able to deliver the expected service when the faults of that set occur. Fault tolerance is implemented mainly by error detection and system recovery.

— **Fault evasion:** During system execution, it is sometimes possible to monitor the system behaviour and detect some abnormal behaviour that suggests that some component is likely to fail or is under some strain. Fault evasion is the use of some compensating action is used to avoid some fault or its consequences.

Avizienis et al. [21] present similar techniques for achieving dependability: fault prevention that can be compared to fault avoidance; fault tolerance; fault elimination that can be compared to fault removal; and fault forecasting that is similar to fault evasion but refers to the way to estimate the current number of faults, any future incidence and the likely consequences of faults. Fault forecasting is conducted by performing an evaluation of the system's behaviour with respect to fault occurrence or activation.

Safety analysis is only concerned with faults that compromise safety and also it considers what happens if the system environment changes somehow in an extreme way and some parts of the system do not behave as planned. Security analysis is similar to safety analysis but focuses on malicious attacks or faults that compromise security. The traditional definition of security is given as a composite notion, namely the combination of confidentiality, integrity and availability, where all these three concepts are attributes of dependability. Safety is also an attribute of dependability and denotes the absence of catastrophic consequences on the user or environment.

It should also be appreciated that the desired system reaction to a fault may be different depending on whether the system is safety-critical. While normally in safety-critical systems reactions to detected faults tend to bring the system to a halt (if a fault occurs in a nuclear plant, for example, the shutdown of the system is normally recommended), in non safety-critical systems it is not always desirable to halt the system, as this could be the intent of a malicious attacker (e.g., an attack that is aimed at causing denial of service).

Fault tolerance and evasion are the dependability and security addressing techniques which relate to the deployment and operational life of a system and hence relevant to our survey. In Section 6, we have discussed techniques for monitoring the operation of a system that can broadly address fault evasion. Thus, in the rest of this section we focus on fault tolerance and discuss it further.

5.2.1. *Fault-tolerance*

Fault tolerance aims to ensure that a correct system service is delivered, even when active faults exist. Fault tolerance is implemented by *error detection* and *recovery* [21].

Error detection identifies error signals or messages in the system. A latent error is an error that is present but not detected. Two classes of error detection techniques exist [21]:

— Concurrent error detection, which happens during service delivery,

— Pre-emptive error detection that occurs which the service is suspended. It checks for latent errors and dormant faults.

Error recovery aims to convert a system state that contains one or more errors or faults, into a state that does not contain any detected errors or faults that can be re-activated. The main activities of recovery are error and fault handling.

Error handling removes errors from the system state and can be realised by the following means [21]:

— **Rollback** – Rollback is an error handling approach which returns a system back to a state saved before an error was detected. This state is referred to as a *checkpoint*. This approach is also known as *backward recovery* (see section 5.5). Checkpointing is a popular technique for reducing the recovery time from a fault and much research has been devoted to analysing checkpoint schemes and determining optimal checkpoint placement strategies [314]. The existence of optimal checkpoint placement strategies is significant as the frequency of and the intervals between checkpoints affect the system's execution time. Also, there is a trade-off between the re-processing time (i.e, the time taken for processing after a fault has occurred) and the overhead of checkpointing that must be considered [315].

- **Rollforward –** Rollforward is an error handling approach which attempts to construct a new system state without the detected errors. This approach is known as "forward recovery" (see section 5.5). Fail-safe behaviour may be seen as a rollforward technique. A fail-safe system is "a system whose failures are, to an acceptable extent, all minor ones" [128].
- **Fault masking –** This is a form of recovery that uses sufficient redundancy to allow for recovery without explicit error detection [21]. Fault masking techniques hide the effects of failures through the means that redundant information outweighs the incorrect information. An example of fault masking is majority voting, where the idea is to take a majority vote on a calculation replicated N times.

Fault handling prevents detected faults from being re-activated and, according to [21], consists of:

- **fault diagnosis** which identifies and records the causes of errors in terms of both location and type;
- **fault isolation** which excludes, physically or logically, the faulty component of a system from further involvement in service delivery and thus makes the fault dormant (a fault is dormant when it cannot produce an error) [21];
- **system reconfiguration** which switches the responsibility for system functions from faulty components to spare components or reassigns functions between non-failed components; and
- **system reinitialisation** which checks, updates and records the new configuration of a system and updates its tables and records.

Fault handling is usually followed by corrective maintenance in which faults that have been identified and isolated by fault handling are removed. The main difference between maintenance and fault tolerance is that the former requires an external agent to be involved in the process.

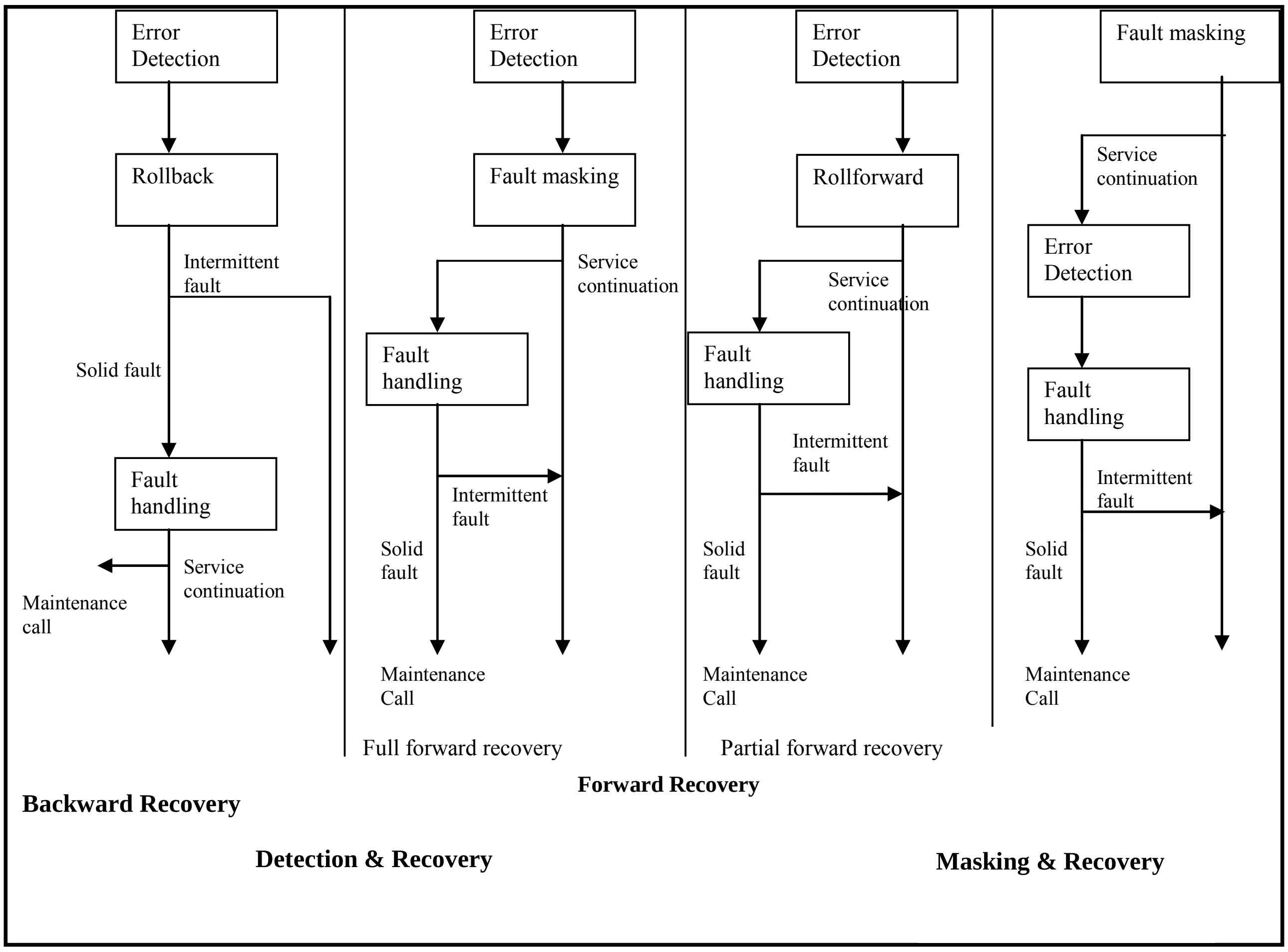


Figure 5.1 – Some strategies for implementing fault tolerance [316]

Figure 5.1 illustrates four typical schematic examples of strategies for implementing fault tolerance which have been suggested by Avizienis et al. [316]. Some points that Avizienis et al. [316] suggest are noteworthy:

— Rollback and rollforward are not mutually exclusive. Rollback can be undertaken first and subsequently, if the error still persists, rollforward can also be performed.

— Intermittent faults are faults that occur periodically. They can be identified by error handling (if an error re-occurs then it is not intermittent) or via fault diagnosis when rollforward is used. Isolation and reconfiguration are not required.

— After error detection error handling can be skipped and fault handling may directly follow.

The choice of error detection, fault handling and error handling techniques depends on the underlying assumptions about faults. The types of faults that can actually be tolerated depend on the fault assumptions made. For example, a system might choose different fault-tolerance mechanisms for dealing with deliberate malicious faults than the mechanisms used for dealing with natural faults.

Fault tolerance techniques have also been distinguished depending on whether they apply to single or multiple versions of software systems. More specifically, Torres-Pomales [317] distinguishes between *single version* and *multi-version fault tolerance techniques*. In the former of these types redundancy is introduced to a single version of a piece of software in order to detect and recover from faults. In the latter type, more than one version of a piece of software are executed in parallel or in sequence to avoid and recover from faults.

According to Torres-Pomales [317] single version fault tolerance techniques include:

— **Software structure and actions:** These techniques are concerned with the software architecture. For example, decomposition techniques can be used to break up a system into components which are used to achieve fault tolerance. Partitioning techniques aimed at providing isolation between functionally independent modules which may also be applied to achieve fault tolerance are classified into this category.

— **Error detection:** Error detection techniques are concerned with the introduction of capabilities for detecting external errors (from information passed to it from other system components) and internal errors into a system.

— **Exception handling:** These techniques support the interruption of normal system operation to deal with abnormal responses.

— **Checkpointing and restart:** This is a recovery mechanism (there are only a few in single version software). A restart can be of two types: static and dynamic. Static corresponds to forward recovery and dynamic corresponds to backward recovery. Checkpoints can be created at fixed intervals or at particular points during execution determined by some optimising rule. The advantage of these checkpoints is that they are based on states that are created during execution, and thus can be used to allow forward progress of execution without having to lose all the work produced up to the when the error was detected.

— **Process pairs:** The recovery mechanism used here is checkpoint and restart. A process pair uses two identical versions of the software that run on separate processors, which are labelled primary and secondary. Firstly, the primary processor actively processes the input and creates the output while generating checkpoint information that is sent to the secondary process (backup). When an error is detected, the secondary processor loads the last checkpoint as its initial state and takes over the role of the primary processor. The faulty processor goes offline and performs some diagnostic checks. After the primary processor is repaired, it takes on the role of the secondary processor and receives checkpoints from the primary. The advantage of this technique is that the service is delivered continuously even with the detection of a failure in the system.

— **Data diversity:** This technique aims to increase the effectiveness of checkpoint and restart by using different input re-expressions (input is changed) on each entry. The objective of each entry is to generate output results that are either exactly the same or semantically equivalent. Three data diversity models are presented: input data re-expression, input re-expression with post-execution adjustment, re-expression via decomposition and recombination. Data diversity can be used together with Process Pairs and also with the multi version fault tolerance techniques.

According to Torres-Pomales [317], multi version fault tolerance techniques include:

— **Recovery blocks:** This technique combines the checkpoint and restart approach and applies them to multiple versions of a software component. Checkpoints are created before a version of a component is executed and are used to recover from an error when this version of the component fails. The essence of the technique is that a different version of the component will be tried when an error is detected.

- **N-version programming:** In this technique, multiple versions of a component are designed to satisfy the same basic requirements and the decision of output correctness is based on the comparison of all the outputs of these components. The difference between this and the recovery blocks approach is that the former uses a generic decision algorithm (usually a voter) to select the correct outputs. N-version programming requires more effort for developing the system but it is not more complex than building a single version.
- **N self-checking programming:** This technique combines multiple software versions with structural variations of Recovery Blocks and N-Version Programming. In this case, the versions and the acceptance tests are developed independently from common requirements. Also, separate acceptance tests are used for each version.
- **Consensus recovery blocks:** This technique combines N-Version Programming and Recovery Blocks to enhance reliability. It uses a decision algorithm similar to N-version programming as a first layer of decision. If a failure is detected in this layer, then a second layer using acceptance tests is invoked. This is a more complex model and because of its complexity it could result in a less reliable system due to the introduction of errors.
- **t/(n-1) – variant programming:** This technique uses an architecture that consists of n variants and uses the t/(n-1) diagnosibility measures to isolate faulty units to a subset of size at most (n-1) assuming there are at most t faulty units. Therefore, at least one non-faulty unit exists such that its output is correct and can be used as the result of the computation module. It can potentially tolerate multiple dependent faults among the versions.

5.3. Recovery for intrusion-tolerant systems

Intrusion tolerance has emerged in the past decade and gained momentum recently [152]. It is the notion of dealing with a wide set of faults, including intended and malicious faults, which may lead to system failure if nothing is done to counteract their effect on the system [244]. In other words, it’s a tolerance paradigm in security that:

- Assumes that attacks on the system can happen, and hence malicious or other faults occur;
- Faults generate errors which compromise component-level security;
- Error processing mechanisms are used to prevent security failure.

A complete approach combines tolerance with prevention, removal, forecasting and all the typical dependability actions. Some researchers present intrusion-tolerant systems as being the "new era of survivability" [152, 197]. They are considered as being the third generation of security systems that shift the security paradigm from simply warding off intruders at all costs, to a more cost-effective and affordable approach of combining prevention, detection and tolerance.

Verissimo et al. [244] discuss two error processing mechanisms in order to recover from intrusions:

- **Processing the errors deriving from intrusions.** The typical error processing mechanisms used in fault tolerance for the IT view are: error detection, error recovery and error masking. We have already described these briefly in Section5.2.1.

— **Intrusion detection mechanism.** Classic intrusion detection systems can be divided into two types: (a) behaviour based (or anomaly) detection systems, and (b) knowledge based (or misuse) detection system. Behaviour based detection systems are characterised by the fact that they need no knowledge about specific tasks. Systems of this type are only provided with knowledge of the normal behaviour of the monitored system and do not require any additional knowledge of attack signatures to guide monitoring. Knowledge based detection systems, on the other hand, rely on such knowledge. In such systems, when an activity matches an attack signature in the knowledge base, an alarm containing diagnostic information about the cause is generated. The main drawbacks of behaviour based intrusion detection systems are that they may generate false alarms if the usage of the system is not predictable with time and they cannot provide diagnostic information is with the alarm. Their main disadvantage of knowledge based intrusion detection systems is that they cannot detect attacks which are not store in their knowledge base, i.e. unknown or new attacks.

Combinations of intrusion detection and automated recovery mechanisms have recently been under investigation in the context of specific architectures, such as in the Willow architecture [145], the MAFTIA project [6]. Cukier et al. [76] and Connelly and Chien [67].

5.4. Recovery in survivable distributed systems

Survivability is a new property of dependability and since it is also a new research area, a precise definition of it is still under debate. According to the most popular definition in the literature, survivability is defined as "the ability of a system to fulfil its mission, in a timely manner, in the presence of attacks, failures and accident" [91, 92, 93, 121, 51, 146]. In this definition, the term "system" is used in a broad sense, including networks as well as large scale systems and the term "mission" signifies a set of abstract requirements or goals. Survivability is a property implying that a system can deliver essential services and maintain essential properties such as performance, security, reliability, availability and modifiability despite the presence of intrusions and – compared to traditional security measures that require central control or administration – survivability aims to address unbounded network environments.

In his survey of IT systems survivability, Tarvainen [233], has summarised the different definitions of survivability that have been given in the literature and points out that the definition of survivability depends on the domain. Despite some similarities between them, survivability is different from fault tolerance [233]: fault tolerance is a mechanism for achieving certain dependability properties while survivability is a dependability property. Moreover, describing a system as fault tolerant is a comment about how the system was designed.

	Definition	Domain	Reference
1	“Survivability is the degree to which essential functions are still available even though some part of the system is down.”	IT systems in general	[83]

2	"Survivability is a property of a system, subsystem, equipment, process or procedure that provides a defined degree of assurance that the names entity will continue to function during and after a natural or man-made disturbance. Note: Survivability must be qualified by specifying the range of conditions over which the entity will survive the minimal acceptable level or post-disturbance functionality and the maximum acceptable outage duration."	Telecommunication Systems	[237]
3	"Survivability is the ability of a network computing system to provide essential services in the presence of attacks and failures and recover full services in a timely manner."	Network Computing Systems	[91]
4	"Survivability is the capability of a system to fulfil its mission, in a timely manner, in the presence of attacks, failures or accidents."	Critical and defence systems	[92, 50, 93, 121, 51, 146, 143]
5	"Survivability is the ability [of a system] to continue to provide service, possibly degraded or different, in a given operating environment when various events cause major damage to the system or its operating environment."	Critical and defence systems	[144, 147]

Table 5.4 – Definitions of Survivability [233]

Survivability is also sometimes viewed as being the same as security. A survivable system must be able to survive from a malicious attack, hence survivability involves security. For example, a system that consists of some security mechanisms, such as passwords and encryptions, may still be vulnerable as it might fail if the server or network link dies. Two aspects of survivability are identified by Tarvainen [233]: survival by protection and survival by adaptation. Survival by protection involves the use of security mechanisms, such as access control and encryption, for protecting applications from harmful, accidental and malicious changes in the environment. Survival by adaptation consists of monitoring and changing the Quality of Service available to applications.

A number of survivability architectures, such as ITDOS [210] and SABER [138], have been defined for designing systems specifically to deal with certain faults. However, these architectures are not mature enough for practical use. Moreover, in this section we are primarily interested in recovery models for survivable distributed systems. Park and Chandramohan [198] present three Recovery Models: Static, Dynamic and Hybrid Recovery Models. Detailed schemas for the static and the dynamic models are also described [198].

— **Static Recovery Model:** This recovery model is based on redundant servers that are prepared before execution, to provide critical services continuously in a distributed client-server environment. The dynamic reconfiguration approach is associated with this model even though it uses the term "dynamic" because the components are generated before execution. Redundancy in different machines or domains enhances survivability because the replaced server can be running in an unaffected area. For example, if the redundant servers are distributed in different network places, then in the event of network failures, the servers can be recovered in different environments. In the case where a failure occurs within a server, it is not effective to replace the server with an identical copy because identical components in the same environment will still be vulnerable.

— **Dynamic Recovery Model:** This recovery model replaces components, which cause failures, contained failures, or are under attack, dynamically by generating components on the fly. These components are deployed at runtime as and when they are required. Moreover, this model can replace infected components by immunised components, thus providing more robust services than that of the static model. Immunised components are components that are not vulnerable to the same type of failure or attack as the infected component.

— **Hybrid Model:** This recovery model combines features of both the static and the dynamic models in order to improve on the disadvantages of both. As shown in Table 5.5, the disadvantage of the dynamic model is with regards to service downtime. This could range from seconds to a few minutes, which suggests that there will be no service available for clients during the recovery period. Alternatively, the disadvantage of the static model concerns resource efficiency, adaptation and robustness. The main weakness of the hybrid model is its more complex to implement than the other two models.

Park and Chandramohan's [198] have compared the above survivability models. A summary of this comparison is shown in Table 5.5.

	Static Recovery Model	Dynamic Recovery Model	Hybrid Model
Simplicity	Higher	Medium	Lower
Resource Efficiency	Lower	Higher	Medium
Adaptation	Pre-fixed	Dynamic	Pre-fixed & Dynamic
Service Downtime	Shorter	Longer	Shorter
Immunization	Environments	Environments & Components	Environments & Components
Robustness	Lower	Medium	Higher

Table 5.5 – Comparison of the three survivability models [198]

5.5. Information warfare defence

Certain organisations, such as defence and civil, depend heavily on their information systems and networks, to the extent that a malicious attack could have devastating effects. Much attention has been given to the prevention and detection of attacks. However, hacker attacks have proved that protective mechanisms are not infallible. If a successful attack is made on a system, the system must be able to identify the attack and respond in a way that maintains system availability of critical functions and allows recovery of capabilities to proceed. Also, any damage that is incurred must be contained. For example, if a cryptographic key for one file is recovered, an attacker should not be allowed to read all of the files on hard disc [318]. Another example given by Schneier [318] concerns smart cards. If an attacker performs reverse-engineering on a smart card to obtain its secrets, s/he should not be able to obtain information that would allow him/her to break other smart cards in a system.

The defender aims to anticipate and block possible attacks, to detect and respond in a way that limits the damage and ensure that critical activities are functioning, while simultaneously the system is in the process of recovery. Jajodia et al. [130] identify the defender's cycle of activities:

- **Prevention**: Protective measures are put into place by the defender.
- **Attack detection:** The defender observes symptoms of a problem that determines that an attack is going to happen or has happened. The defender collects information in order to diagnose whether symptoms are due to a legitimate activity or not.
- **Damage assessment and containment:** The defender determines the extent of damage to the system by examining it. Also, it takes immediate action to ensure the attacker is excluded from the system and to contain the problem.
- **Recovery:** The defender may reconfigure the system to allow it to operate in a degraded mode while recovery proceeds. For example, it might need to cut back on some non-critical functions in order to deal with the critical functions. The defender then recovers any corrupted or lost data, and reinstalls any missing functions in order to restore normal operation.
- **Fault treatment:** Weaknesses in the systems that the attacks uncovered are examined and steps are taken to ensure this attack is not repeated. This phase in the cycle relates to both prevention and reaction.

In the field of fault-tolerance, two types of errors are identified: anticipated errors and those that are unanticipated [14]. For anticipated errors, the prediction or assessment of damage can be made. This is not true of unanticipated errors. To recover from these errors the following two methods have been defined and discussed by Jajodia et al. [130]. These are the same as rollforward and rollback, but we describe them again in the context of information defence.

- **Forward Recovery:** These methods are usually used to recover from anticipated errors. Since these errors can be predicted, contingency update instructions can be defined or a means of deriving a correct value. If the recovery method is supported by the semantics, compensating transitions can anticipate error scenarios [319]. For some items that are replaced regularly through normal processing, the errors can be corrected simply by waiting for the replacement transactions to occur. There are two limitations with regards to forward recovery methods. Firstly, these methods are system specific. Secondly, the success of these methods depends on how accurate the capability of predicting or assessing the damage from faults is.

— **Backward Recovery:** These methods are used to recover from unanticipated errors. They require that the entire state be replaced by a consistent prior state. This method is not optimal as it requires that the system be halted temporarily. For denial of service attacks, it may be the attacker's intention to halt the system and this could be harmful if the system is halted at a critical moment. Database management systems (DBMS) provide a rich set of recovery methods that mainly depend on backward recovery methods for restoring the database to a consistent state. There are some limitations with backward recovery methods used in DBMS and specification with regards to security attacks. Firstly, suppose that a transaction is aborted, the transaction isolation property supports recovery in the sense that the transaction can be withdrawn without completion, without affecting other transactions. In the case of a malicious attack, the isolation property does not help because the transaction placed by the attacker seems normal to the DBMS and is completed (but creates bad data). Undo/redo logs assist in recovery when the system fails with a number of uncompleted transactions, however not in this case. In the meantime, other transactions might use the bad data created to perform some computations, and store the results in other items (hence creating further bad data). The only general mechanism that is available is backward recovery that will roll the database back to an approved checkpoint. However, the problem with this mechanism is that all other computations undertaken after this checkpoint are also lost.

Three recovery models are presented by Jajodia et al. [130] that can be used to formalise recovery methods:

— **HotStart:** This can be seen as being a mostly forward recovery method. It is appropriate for attacks in which the system must respond transparently to the user. Let's assume that an attacker introduces a corrupted executable at a particular site and use it to initiate an availability, integrity or trust attack. This attack can be dealt by a HotStart model if it satisfies the following two conditions. Firstly, the attack must have been detected early enough so that the damage is limited to the executable. Secondly, an uncorrupted standby (known as hot standby) must be available to take over. Also, the path by which the attacker introduced the corrupt binary must be disabled and the proper binary from the backup store must be restored.

— **WarmStart:** This model should be used when it is difficult to hide all aspects of response to the user. The users are aware of the attack as the system operation is degraded. However, damage must be contained and the main system services must be available, trustworthy and reliable. The level of service depends on the extent of the attack. Some of the functionality might be missing, could be untrustworthy or the information held could be incorrect. Usually, checkpoints are used for quick recovery and audit trails for intercepting the attacker. If an availability attack occurs, a WarmStart would respond in a nontransparent but automatic way to recover the system from confined damage. If a trust attack occurs, a WarmStart response would mean that only certain operations could be trusted while the response to the attack occurs. If an integrity attacks occurs, a WarmStart response means that only certain system functionality is enabled.

— **ColdStart**: This can be seen as being mostly a backward recovery method and is appropriate for most serious attacks. For example, when the attacker manages to halt the delivery of system services. The aim of the ColdStart is to get the system running as soon as possible in a usable, trustworthy and consistent state. Effective CodStarts must be supported by policies and algorithms. Also, compensation for unrecoverable components, such as leaked information that the intruder now knows , is vital.

Jajodia et al. [130] present several additional methods (i.e. not only forward and backward recovery) that could be used to deal with recovery and these are discussed in relation to the recovery models.

- **Redundancy:** This is a key technique for recovery and it means that either a data item is stored redundantly somewhere in the system and retrieved when lost or damaged, or it can be derived from some other elements in the system. Such redundancy take the form of alternative algorithms, backups at geographically distant locations, compensation methods for unrecoverable components, and audit trails for tracking the system usage and access. Redundancy is useful in all three recovery models. For example in the HotStart model, if an attack that has damaged an executable, then a hot standby executable at a different geographic location can take over. For WarmStart recovery, derived data could be of use as these could be labelled with attribute evaluation rules explain how to derive them from other attributes that could be found outside the system. Coldstart models make use of recovery logs.
- **Static partitioning of information elements:** This method pertains to the design of the database and its applications in such a way that transactions only affect data in a single region. Hence, damage caused by an attack can be contained and applications that use other partitions of the database can proceed normally. This design could be impractical for many databases. An alternative solution could be to define borders of regions, identify triggers or generate updates that cross the borders, and limit the conditions under which the data may flow across [130].
- **Versioning:** This concept is borrowed from concurrent engineering. Trees consisting of versions, where each version is a checkpoint between transactions, are maintained to enable a more elegant restoration of a consistent state. If the current database state was found to be corrupted, then a different branch could be adopted. Therefore, this type of versioning is closely linked to states of the database applications.
- **Dynamic partitioning of information elements:** The objective of this method is to use recovery methods to identify data items that can be removed from use, repaired and added back for use dynamically. This is a crucial technique for the HotStart recovery model.
- **Countermeasure transactions:** These are special type of transactions that are designed to repair or detect damage. An attack can be detected by a variety of means. These can be grouped into two categories. Those that are internal to the database, for example, an integrity constraint violation detection via the firing of an action rule in an active database. Those that are external to the database, for example, an alert officer that notices a abnormally high number of aircrafts are scheduled for refuelling at a specific tanker. Also, damage can be repaired using drastic measure such as resetting the database to a prior consistent state or by defining transaction that will eventually overwrite the bad data with good data. The advantage of using countermeasure transactions for recovery is that the power of the transaction model can be used to implement fault tolerance across the entire system.

6. Conclusion

In this deliverable we provided a review of the state of the art in the dynamic verification (aka monitoring) of security and dependability properties, and methods developed to support recovery from violations of such properties when they occur. Our review has also covered static verification techniques and discussed limitations of these techniques that demonstrate the reasons that make dynamic verification a necessary verification instrument for complex and highly interoperable and dynamic systems[2]. We have also provided an overview of different languages and notations that have been used to specify the behaviour of systems and the dependability and security properties that need to be verified against this behaviour statically or dynamically.

As part of our survey, we have also identified a number of open research issues related to dynamic system verification that will inform further research in Activity 4 of the project.

[2] Our survey has covered static verification techniques to a lesser extent than dynamic verification as the former techniques are outside the scope of A4 and are covered by the state of the art reviews that have been conducted by other problem activities in SERENITY

References

[1] Abadi, M. and Gordon, A. D, (1997). “A calculus for cryptographic protocols: The spi calculus”. In Proceedings of the Fourth ACM Conference on Computer and Communications Security, pages 36-47.

[2] Abadi, M. and Rogaway, P, (2001). “Reconciling Two Views of Cryptography (The computational soundness of formal encryption)”. Journal of cryptology.

[3] Abadi, M. and Tuttle, M. R, (1991). “A Semantics for a Logic of Authentication”. Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada, August 1991, pp. 201-216

[4] Abercrombie, P. and Karaorman, M. (2002). “jContractor: Bytecode instrumentation techniques for implementing design by contract in java”. In Electronic Notes in Theoretical Computer Science, volume 70. Elsevier Science Publishers.

[5] Abrial, J. R, (1996). “The B-book: Assigning Programs to Meanings”. Cambridge University Press.

[6] Adelsbach, A., Alessandri, D., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J.C., Powell, D., Randell, B., Riordan, J., Ryan, P., Simmonds, W., Stroud, R., Verissimo, P., Waidner, M. and Wespi, A. (2002). “Conceptual Model and Architecture of MAFTIA”. Project MAFTIA deliverable D21.

[7] Al-Azzoni, I., Down, D. J. and Khedri, R, (2005). “Modeling and Verification of Cryptographic Protocols Using Coloured Petri Nets and Design/CPN,”MOMPES Workshop.

[8] Albern, B. and Schneider, F. B, (1987). “Recognizing safety and liveness”. Distrib. Comput. 2, 117–126.

[9] Alexander, I, (2003). “Misuse Case Help to Elicit Nonfunctional Requirements”. IEE CCEJ.

[10] Alpern, B. and Schneider, F. B, (1985) “Defining liveness”. Information Processing Letters, vol. 21, pp. 181-185.

[11] Aly, S. and Mustafa, K, (2003). “Protocol Verification And Analysis Using Colored Petri Nets”. Technical Report, DePaul University, TR04-003.

[12] Amir, J, (2004). “A Survey of Runtime Verification”. Conference on Automated Verification, University of Toronto, January, 2004

[13] Anderson, R, (1992). “UEPS: A Second Generation Electronic Wallet”. Computer Security ESORICS '92, Springer-Verlag, pp. 411-418.

[14] Anderson, T. and Lee, P. A, (1990). “Fault Tolerance: Principles and Practice”. Springer-Verlag, Wien - New York

[15] Andrieux, A. et al. (2004). “Web Services Agreement Specification”, Global Grid Forum, available from: http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf

[16] Artho, C. and Biere, A, (2005). "Combined Static and Dynamic Analysis". in Proc. AIOOL '05, Paris, France.

[17] Artho, C., Biere, A. and Havelund, K, (2004). "Using block-local atomicity to detect stale value concurrency errors". In Farn Wang, editor, Proc. ATVA '04. Springer.

[18] Artho, C., Havelund, K. and Biere, A, (2003). "High-level data races". Journal on Software Testing, Verification & Reliability (STVR), 13(4).

[19] Artho, C., Schuppan, V., Biere, A., Eugster, P., Baur, M. and Zweimüller, B, (2004). "JNuke: Efficient Dynamic Analysis for Java". In Proc. 16th Intl. Conf. On Computer Aided Verification (CAV 2004), volume 3114 of LNCS, pp. 462–465, Boston, USA. Springer.

[20] Atelier B., v 3.6 http://www.atelierb.societe.com/index_uk.html.

[21] Avizienis, A., Larpie, J. C. and Randell, B, (2000). "Fundamental Concepts of Dependability". In Information Survivability Workshop.

[22] Axelsson, S. (2000). "Intrusion Detection Systems: A Survey and Taxonomy". Technical Report 99-15, Depart. of Computer Engineering, Chalmers University.

[23] Balasubramaniyan, J., Garcia-Fernandez, J.O., Isaco, D., Spaord, E.H. and Zamboni, D, (1998). "An architecture for intrusion detection using autonomous agents". Technical Report Coast TR 98-05, The COAST Project, Dept. of Comp. Sciences, Purdue Univ.,West Lafayette, IN, 47907, USA.

[24] Baldwin, R. W, (1990), "Naming and Grouping Privileges to Simplify Security Management in Large Database", In Proceedings IEEE Computer Society Symposium on Research in Security and Privacy, pp. 184–194.

[25] Bandara, A. K., Lupu, E. C. and Russo, A, (2003). "Using event calculus to formalise policy specification and analysis". Policies for Distributed Systems and Networks Proceedings, POLICY 2003, pp. 26- 39.

[26] Baresi, L. and Guinea, S. (2005). "Dynamo: Dynamic Monitoring of WS-BPEL Processes", ICSOC 05, 3rd International Conference On Service Oriented Computing, Amsterdam, The Netherlands.

[27] Baresi, L. and Guinea, S. (2005). "Towards Dynamic Monitoring of WS-BPEL Processes". ICSOC 05, 3rd International Conference On Service Oriented Computing, Amsterdam, The Netherlands.

[28] Baresi, L., Guinea, S. and Plembani, P, (2005). "Using WS-Policy in Service Monitoring". TES 05, 6th VLDB Workshop on Technologies for E-Services,Trondheim, Norway.

[29] Barnett B. and Vu, D. N, (1997). "Vulnerability assessment and intrusion detection with dynamic software agents". In Proceedings of the Software Technology Conference.

[30] Barnett, M. and Schulte, W, (2001). "Spying on Components: A Runtime Verification Technique". In Proceedings of OOPSLA 2001 Workshop on Specification and Verification of Component Based Systems, Tampa, FL, USA.

[31] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J. C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saibi, A. and Werner, B, (1998). "The Coq Proof Assistant Reference Manual - Version 6.2". *INRIA*, Rocquencourt.

[32] Barringer, H., Goldberg, A., Havelund, K. and Sen, K, (2004). “Rule-Based Runtime Verification”. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’04), LNCS 2937, Springer, pages 44-57.

[33] Bartetzko, D., Fischer, C., Moller, M., and Wehrheim, H, (2001). “Jass – Java with assertions”, In Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01. Published in Electronic Notes in Theoretical Computer Science, K. Havelund and G. Rosu (eds.), 55(2).

[34] Bauer, L., Ligatti, J. and Walker, D, (2002). “More enforceable security policies”, In Foundations of Computer Security, Copenhagen, Denmark.

[35] B-Core (UK) Ltd, http://www.b-core.com/.

[36] Bernstein, P.A., Hadzilacos, V., and Goodman, N, (1987). “Concurrency control and recovery in database systems”. Addison-Wesley Publishing Company

[37] Bieber, P. (1990). “A logic of communication in hostile environment”. In Proceedings of the IEEE Computer Security Foundations Workshop.

[38] Blum, M. and Micali, S. (1984). “How to generate cryptographically strong sequences of pseudo-random bits”. SIAM Journal of Computing, 13(4):850--864

[39] Bolignano, D, (1996). “An approach to the formal verification of cryptographic protocols”. In 3rd ACM Conference on Computer and Communications Security, pages 106—118.

[40] Boreale, M., De Nicola, R., and Pugliese. R. (2000). “Process Algebraic Analysis of Cryptographic Protocols”. Proc. of 13th FORTE / 20th PSV, Kluiver.

[41] Brackin, S. H. (1996). “A HOL Extension of GNY for Automatically Analyzing Cryptographic Protocols”. Proc. of Computer Security Foundations Workshop. IEEE Press.

[42] Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Visser, W. and Washington, R, (2004). “Experimental Evaluation of Verification and Validation Tools on Martian Rover Software”. Formal Methods in System Design, 25(2).

[43] Brewer, D. F. C. and Nash, M. J, (1989). “The Chinese Wall Security Policy”. Proceedings IEEE Computer Society Symposium on Research in Security and Privacy, pp. 215–228.

[44] Brisset, P. (2000). “A Case Study in Java Software Verification”. Appeared in Workshop on Security, Middleware, and Languages, Stockholm.

[45] Brörkens, M. and Möller, M, (2002). “Dynamic event generation for runtime checking using the JDI”, In Havelund, K. and Rosu, G., editors, Proceedings of the Federated Logic Conference Satellite Workshops, Runtime Verification, Copenhagen, Denmark. Electronic Notes in Theoretical Computer Science 70(4).

[46] Brörkens, M. and Möller, M, (2002). “Jassda trace assertions, runtime checking the dynamic of java programs”. In Schieferdecker, I., König, H., and Wolisz, A., editors, Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems, Berlin, Germany, pages 39-48.

[47] Burns J. and Mitchell C, (1990). “A Security Scheme for Resource Sharing over a Network”. Computers and Security, Vol. 19, 67-76.

[48] Burrows, M., Abadi, M., and Needham, R. M, (1990). “A Logic of Authentication”. ACM Transactions on Computer Systems, Vol. 8, No. 1, pp. 18-36.

[49] Buttyan, L. (1999). “Formal Methods in the Design of Cryptographic Protocols (state of the Art)”. Technical Report, Swiss Federal Institute of Technology, Lausanne, Switzerland.

[50] Byon, I, (2000). “Survivability of the U.S. Electric Power Industry”, Master’s Thesis, Carnegie Mellon University, Information Networking Institute.

[51] Caldera, J, (2000). “Survivability Requirements for the U.S. Health Care Industry”, Master’s Thesis, Carnegie Mellon University, Information Networking Institute.

[52] Campbell, E. A., Safavi-Naini, R. and Pleasants, P. A, (1992). “Partial Belief and Probabilistic Reasoning in the Analysis of Secure Protocols”. In Proceedings 5th IEEE Computer Security Foundations Workshop, pages 84-91. IEEE Computer Society Press.

[53] Capra, L., Emmerich, W. and Mascolo, C, (2001). “Reflective middleware solutions for context-aware applications”. In Yonezawa, A., Matsuoka, S., eds.: Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan. LNCS 2192, AITO, Springer-Verlag, pp.126–133.

[54] Capra, L., Emmerich, W. and Mascolo, C, (2003) “CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications”. In IEEE Transactions on Software Engineering, 29(10), pp.929-945.

[55] CCITT (1998), The directory authentication framework. CCITT Recommendation X.509.

[56] CEN/ISSS (2003), Digital Rights Management Report.

[57] Chang, E., Pnueli, A., Manna, Z. (1994). “Compositional Verification of Real-Time Systems”. Proc. 9'th IEEE Symp. On Logic In Computer Science, 1994, pp. 458-465.

[58] Chapman, D. B. (1992). “Network (In)Security Through IP Packet Filtering”. In USENIX Security Symposium III Proceedings, pages 63-76. USENIX Association.

[59] Chen, F. and Rosu, G, (2003). “Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation”. In Electronic Notes in Theoretical Computer Science 89 No. 2, Published by Elsevier Science B.V.

[60] Chen, F. and Rosu, G, (2005). “Java-MOP: A Monitoring Oriented Programming Environment for Java”. Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05).

[61] Cholewka, D. G., Botha, R. A., Eloff, J. P, (2000). “A context-sensitive access control model and prototype implementation”, In Information Security for Global Information Infrastructures: IFIP TC 11 Sixteenth Annual Working Conference on Information Security. Beijing, China. pp. 341-350.

[62] Clarke, E.M., Grumberg, O., and Peled, D. (1999) “Model Checking”. MIT Press

[63] Clavel, M., Durán, F. J., Eker, S., Lincoln, Martí-Oliet, N., Meseguer, J. and Quesada, J. F. (1999), “The Maude System”. In Proceedings of the 10th International Conference on Rewriting Techniques

and Applications (RTA-99), Vol. 1631 of LNCS. Trento, Italy, pp. 240–243, Springer-Verlag. System description.

[64] Cohen, D., Feather, M., Narayanswamy K. and Fickas, S, (1997). “Automatic Monitoring of Software Requirements”. In Proc. of the 19th Int. Conf. on Software Engineering.

[65] Cohen, G., Chase, J. and Kaminsky, D, (1998). “Automatic Program Transformation with JOIE”. In Proceedings of the 1998 USENIX Annual Technical Symposium.

[66] Cohen, S. (1999). “Jtrek, Developed by Compaq”. http://www.compaq.com/java/download/jtrek.

[67] Connelly, K., and Chien, A. A. (2002). “Breaking the Barriers: High Performance Security for High Performance Computing”. In Proc. Of New Security Paradigms Workshop.

[68] Corbett, J., Dwyer, M., Hatcliff, J. and Robby (2001). “Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language”. KSU CIS Technical Report 2001-04.

[69] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S. and Zheng, H. (2000). “Bandera: Extracting Finite-state Models from Java Source Code”, in Proceedings of the 22nd International Conference on Software Engineering, June.

[70] Corradi, A., Montanari, R. and Tibaldi, D. (2004). “Context-Based Access Control Management in Ubiquitous Environments”, Network Computing and Applications, Third IEEE International Symposium on (NCA'04), pp. 253-260.

[71] Cotroneo, D., Graziano, A., and Russo, S, (2004). “Security requirements in service oriented architectures for ubiquitous computing”. In Proceedings of the 2nd Workshop on Middleware For Pervasive and Ad-Hoc Computing. ACM Press, New York, NY, 172-177.

[72] Cousot, P. and Cousot, R. (1977). “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In Proc. Symp. Principles of Programming Languages. ACM Press.

[73] Covington, M. J., Moyer, M. J. and Ahamad, M. (2000). “Generalized role-based access control for securing future applications”. In 23rd National Information Systems Security Conference, Baltimore, MD.

[74] Crazzolara, F. and Winskel, G. (2001). “Petri Nets in Cryptographic Protocols”. IEEE-IPDPS-01, pg. 149.

[75] Crosbie, M. and Spafford, E. H. (1995). “Defending a Computer System Using Autonomous Agents” Technical Report CSD-TR-95-022.

[76] Cukier, M., Lyons, J., Pandey, P., Ramasamy, H. V., Sanders, W. H., Pal, P., Webber, F., Schantz, R., Loyall, J., Watro, R., Atighetchi, M. and Gossett, J. (2001). “Intrusion Tolerance Approaches in ITUA”. FastAbstract in Supplement of the 2001 International Conference on Dependable Systems and Networks, July 1-4, Göteborg, Sweden.

[77] Damianou, N., Dulay, N., Lupu, E. C. and Sloman, M. S. (2001), “The Ponder Policy Specification Language”, presented at Policy 2001: Workshop on Policies for Distributed Systems and Networks, Bristol, UK.

[78] d'Amorim, M. and Havelund, K. (2005). "Event-based runtime verification of java programs", In Proceedings of the Third international Workshop on Dynamic Analysis (St. Louis, Missouri, May 17 - 17, 2005). WODA '05. ACM Press, New York, NY, 1-7.

[79] Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). "Goal-Directed Requirements Acquisition" Science of Computer Programming, 20, pp. 3-50.

[80] David, P. C., Ledoux, T. and Bouraqadi-Saadani, N. M. N. (2001). "Two-step weaving with reflection using AspectJ". in OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.

[81] Denning, D. (1987). "An Intrusion-Detection Model", IEEE Transactions on Software Engineering, Vol. SE-13, No. 2, pp. 222-232.

[82] Desai, N, (2003). "Intrusion Prevention Systems: the Next Step in the Evolution of IDS", SecurityFocus, http://www.securityfocus.com/infocus/1670.

[83] Deutsch, M. S. and Willis, R. R. (1988). "Software Quality Engineering: A Total Technical and Management Approach" Englewood Cliffs, NJ: Prentice Hall.

[84] Diffie, W., van Oorschot, P., and Wiener, M. (1992) "Authentication and Authenticated Key Exchange", Designs, Codes and Cryptography, 2, 1992, pp.107-125.

[85] Dingwall-Smith, A. and Finkelstein A. (2002), "From Requirements to Monitors by Way of Aspects", Proc. of 1st Int. Conf. on Aspect-Oriented Software Development.

[86] Dobson, J. E. and McDermid, J. A. (1989), "Security Models and Enterprise Models", pp. 1–39.

[87] Dolev, D. and Yao A. (1983). "On the security of public-key protocols". IEEE Transaction on Information Theory 29, 198-208.

[88] Drusinsky, D, (2000). "The Temporal Rover and the ATG Rover", In K. Havelund, J. Penix, and W. Visser, editors, SPIN Model Checking and Software Verification, volume 1885 of LNCS, pages 323–330. Springer.

[89] Eiffel Software. Eiffel language. http://www.eiffel.com/.

[90] Eilenberg, S. (1974). "Automata, Languages, and Machines". Volume A. Academic Press, Inc., New York, NY.

[91] Ellison, R. J., Fischer, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. and Mead, N. R. (1997). "Survivable Network Systems: An Engineering Discipline", Technical Report CMU/SEI-97-TR-013, Carnegie Mellon University.

[92] Ellison, R. J., Fischer, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. and Mead, N.R. (1999). "Survivability: Protecting your Critical Systems". IEEE Internet Computing, CERT Coordination Centre Software Engineering Institute, pg 55-63.

[93] Ellison, R. J., Fischer, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. and Mead, N. R. (1999). "An Approach to Survivable Systems". Technical Report, CERT Coordination Centre, Carnegie Mellon University.

[94] Emmerich, W, (2000). "Software Engineering and Middleware: A Roadmap". In The Future of Software Engineering - 22nd Int. Conf. on Software Engineering (ICSE2000), pages 117–129, ACM Press.

[95] Feather, M. and Fickas, S, (1995). "Requirements Monitoring in Dynamic Environments". In Proc. of Int. Conf. on Requirements Engineering.

[96] Feather, M. S., Fickas, S., van Lamsweerde, A. and Ponsard, C, (1998). "Reconciling System Requirements and Runtime Behaviour". Proc. of 9th Int. Work. on Software Specification & Design.

[97] Ferraiolo, D. and Kuhn, D. R, (1992). "Role-Based Access Control", Proceedings of the NIST-NSA National (USA) Computer Security Conference, pp. 554–563.

[98] Firesmith, D. G, (2003). "Analyzing and Specifying Reusable Security Requirements". Eleventh International IEEE Conference on Requirements Engineering (RE'2003) Requirements for High-Availability Systems (RHAS'03) Workshop, Monterey, California.

[99] Gaardner, K. and Snekkenes, E, (1991). "Applying a Formal Analysis Technique to the CCITT X.509 Strong Two-Way Authentication Protocol". Journal of Cryptology, 3(2):81—98.

[100] Genrich, H.-J. (1987). "Predicate/Transition Nets". LNCS 254, Springer Verlag.

[101] Giannakopoulou, D. and Havelund, K. (2001). "Automata-Based Verification of Temporal Properties on Running Programs", In Proceedings of International Conference on Automated Software Engineering (ASE'01), pages 412–416. ENTCS. Coronado Island, California.

[102] Goldberg, A. and Havelund, K. (2003). "Instrumentation of Java Bytecode for Runtime Analysis", In Proc. Formal Techniques for Java-like Programs, volume 408 of Technical Reports from ETH Zurich, Switzerland. ETH Zurich.

[103] Goldwasser, S. and Micali, S. (1984). "Probabilistic Encryption". Journal of Computer and System Sciences 28, 270—299

[104] Gong, L., Needham, R. and Yahalom, R, (1990). "Reasoning about Belief in Cryptographic Protocols". Proc. 1990 IEEE Symp. on Security and Privacy (Oakland, California), pp. 234-248.

[105] Gordon, M. and Melhams, T. F. (1993). "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic". Cambridge University Press.

[106] Grimes, R. (2004). "Authenticode", Microsoft Corporation TechNet, Microsoft Authenticode Reference Guide.

[107] Gritzalis, S., Katsikas, S. and Gritzalis, D. (2003). "Computer Network Security", Papasotiriou Publishers.

[108] Groce, A. and Visser, W. (2002). "Model checking Java programs using structural heuristics", In Proc. of the 2002 Int. Symp. On Software Testing and Analysis, pages 12–21.

[109] Guimaraes, J., Boucqueau, J. M. and Macq, M. (1996), "OKAPI: a Kernel for Access Control to Multimedia Services Based on Trusted Third Parties". Proc. ECMAST 96, Louvain-laNeuve, Belgium, 783—798.

[110] Gurevich, Y. (1993). "Evolving algebras: An attempt to discover semantics". In G. Rozenberg and A. Saloma, editors, Current Trends in Theoretical Computer Science, pages 266--292. World Scientific.

[111] Gurevich, Y., Schulte, W., Campbell, C. and Grieskamp, W. (2001). "The Abstract State Machine Language". The Abstract State Machine Language, Microsoft Corporation.

[112] Haley, C. B., Laney, R. C., and Nuseibeh, B. (2004). "Deriving security requirements from crosscutting threat descriptions". In Proceedings of the 3rd international Conference on Aspect-Oriented Software Development, AOSD '04. ACM Press, New York, NY, 112-121.

[113] Hammond J., Rawlings R., Hall A. (2001). "Will It Work". Proceedings 5th IEEE International Symposium on Requirements Engineering

[114] Hatcliff, J. and Dwyer, M, (2001). "Using the Bandera tool set to model-check properties of concurrent Java software". In CONCUR 2001, LNCS 2154, pages 39–58.

[115] Hatcliff, J., Corbett, J. C., Dwyer, M. B., Sokolowski, S. and Zheng, H. (1999). "A formal study of slicing for multi-threaded programs with JVM concurrency primitives". In Proceedings of the 6th International Static Analysis Symposium (SAS'99).

[116] Havelund, K. and Rosu, G. (2001). "Monitoring Java Programs with Java PathExplorer". In Proceedings of the 1st International Workshop on Runtime Verification (RV'01) [1], pages 97–114.

[117] Havelund, K. and Rosu, G. (2001). "Monitoring Programs using Rewriting". In Proceedings of International Conference on Automated Software Engineering (ASE'01), pages 135–143. Institute of Electrical and Electronics Engineers. Coronado Island, California.

[118] Havelund, K. and Rosu, G. (2002). "Synthesizing Monitors for Safety Properties". In Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), volume 2280 of LNCS, pages 342–356. Springer. Extended version to appear in the journal: Software Tools for Technology Transfer, Springer, 2004.

[119] Havelund, K. and Roşu, G. (2004). "An Overview of the Runtime Verification Tool Java PathExplorer", Form. Methods Syst. Des. 24, pp.189-215.

[120] Heberlein, T., Dias, G., Levitt, K., Mukherjee, B., Wood, J. and Wobler, D. (1990). "A Network Security Monitor", Proceedings IEEE Symposium on Research in Computer Security and Privacy.

[121] Hiltunen, M. A., Schlichting, R. D., Ugarte, C. A. and Wong, G. T. (2000). "Survivability through Customization and Adaptability: The Cactus Approach", DARPA Information Survivability Conference and Exposition, pages 294-307.

[122] Hirschfeld, R. and Kawamura, K. (2004). "Dynamic service adaptation", in Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04), Tokyo, Japan.

[123] Hoare, C. (1985-2004). "Communicating Sequential Processes", electronic version of Communicating Sequential Processes, first published in 1985 by Prentice Hall International, http://www.usingcsp.com/cspbook.pdf.

[124] Holzmann, G. J. and Smith, M. H. (1997). "The model checker SPIN". IEEE trans. SE, 23(5), pp. 279–295.

[125] Hulsebosch, R. J., Salden, A. H., Bargh, M. S., Ebben, P. W. and Reitsma, J. (2005). “Context sensitive access control”. In Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies, SACMAT '05, ACM Press, New York, NY, 111-119.

[126] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj.

[127] Iannella, R. (editor) (2002), “Open Digital Rights Language (ODRL)”, Version: 1.1 , http://odrl.net/1.1/ODRL-11.pdf

[128] Isaksen, U., Bowen, J. P. and Nissanke, N. (1996). “System and Software Safety in Critical Systems”. The University of Reading, Whiteknights, United Kingdom, Technical Report RUCS/97/TR/062/A.

[129] Jahanian, J., Rajkumar, R., and Raju, S. (1994). “Runtime monitoring of timing constraints in distributed real-time systems”. Technical Report CSE-TR 212-94, University of Michigan, April 1994.

[130] Jajodia, S., McCollum C. D. and Ammann P. (1999). “Trusted recovery”. Communications of the ACM, Vol. 42, No. 7, pages 71-75.

[131] Janicke, H., Siewe, K., Jones, F., Cau, A. and Zedan, H. (2005). “Analysis and Run-time Verification of Dynamic Security Policies”. AAMAS 05 workshop on Defence Applications of Multi-Agent Systems, Utrecht.

[132] Kailar, R. (1995). “Reasoning About Accountability in Protocols for Electronic Commerce”. In Proceedings of the 14th IEEE Symposium on Security and Privacy, pages 236-250. IEEE Computer Society Press.

[133] Kaler, C. and Nadalin, A. (editors) (2005), “Web Services Security Policy Language (WS-SecurityPolicy)”, http://www-128.ibm.com/developerworks/library/specification/ws-secpol/.

[134] Karaorman, M. and Freeman J. (2004). “jMonitor: Java runtime event specification and monitoring library”, Proceedings of 4th Workshop on Run-time Verification, 2004, available from: http://ase.arc.nasa.gov/rv2004/papers/paper11.pdf

[135] Kazman, R., Yan, H., Garlan, D., Schmerl, B. and Aldrich, J. (2004). “The Recovery of Runtime Architectures”. The Architect, news@sei, Carnegie Mellon University.

[136] Kemmerer, R. A. (1989). “Analyzing Encryption Protocols Using Formal Verification Techniques”. IEEE J. Selected Areas in Comm, 7(4), 448-457.

[137] Kemmerer, R., Meadows, C. and Millen, J. (1994). “Three systems for cryptographic protocol analysis”. Journal of Cryptology, 7(2):79—130.

[138] Keromytis, A. D., Parekh, J., Gross, P. N., Kaiser, G., Misra, V., Nieh, J., Rubestein, D., and Stolfo, S. (2003). “A Holistic Approach to Service Survivability”. Technical Report CUCS-021-03, Department of Computer Science, Columbia University.

[139] Kessler, V. and Wedel, G. (1994). “AUTOLOG -- An Advanced Logic of Authentication”. In Proceedings of the Computer Security Foundations Workshop VII, pages 90--99. IEEE Computer Society Press.

[140] Kiczales, G. and Lamping, J. (1997). “Aspect-oriented programming”. In Mehmet Aksit and Satoshi Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220{242. Springer-Verlag, Berlin, Heidelberg, and New York.

[141] Kiczales, G., Hilsdale,E., Hugunin,J., Kersten, M., Palm, J. and Griswold, W. G. (2001). "An Overview of AspectJ". In Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327–353. Springer-Verlag.

[142] Kim, M., Kannan, S., Lee, I., Sokolsky, O. and Viswanathan, M. (2001). "Java-mac: a run-time assurance tool for java programs". In Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers.

[143] Knight, J. C. and Strunk E. A. (2004). "Achieving Critical System Survivability through Software Architectures Architecting Dependable Systems". (R. de Lemos, C. Gacek, and A. Romanovsky Eds) Springer Verlag.

[144] Knight, J. C., and Sullivan, K. J. (2000). "On the Definition of Survivability". Technical Report CS-TR-33-00, University of Virginia.

[145] Knight, J. C., Heimbigner, D., Wolf, A., Carzaniga, J. H. and Devanbu, P. (2001). "The Willow Survivability Architecture". Proc. of the Fourth Information Survivability Workshop.

[146] Knight, J. C., Strunk, E. A. and Sullivan, K. J. (2003). "Towards a Rigorous Definition of Information System Survivability". DISCEX 2003, Washington DC.

[147] Knight, J. C., Sullivan, K.J., Elder, M.C. and Wang, C. (2000). "Survivability Architectures: Issues and Approaches". DARPA Information Survivability Conference and Exposition (DISCEX 2000), Hilton Head SC.

[148] Ko, C., Ruschitzka, M. & Levitt, K. (1997). "Execution monitoring of security-critical programs in distributed systems: A specication-based approach". In Proceedings of the 1997 IEEE Symposium on Security and Privacy, pages 175-187, Oakland, CA, USA.

[149] Kowalski, R. A. and Sergot, M. J. (1986). "A logic-based calculus of events". New Generation Computing, vol. 4, pp. 67-95.

[150] Kumar, A., Kamik, N. and Chafle, G. (2002). "Context Sensitivity in Role-based Access Control". ACM SIGOPS Operating Systems Review, pp. 53-66.

[151] Kurth, T. A. (2002). "Digital Rights Management: An Overview of the Public Policy Solutions to Protecting Creative Works in a Digital Age". WISE 2002 Intern, Kansas State University.

[152] Lala, J. H. (2000). "Intrusion Tolerant Systems". Pacific Rim International Symposium on Dependable Computing (PRDC 2000), 18-20, Los Angeles, CA, USA. IEEE Computer Society.

[153] Lamsweerde, A. (2004). "Elaborating Security Requirements by Construction of Intentional Anti-Models". In Proceedings of ICSE'04, 26th International Conference on Software Engineering, Edinburgh, May. 2004, ACM-IEEE, pp. 148-157.

[154] Leavens, G., Baker, A. and Ruby, C. (2003). "Preliminary Design of JML: A Behavioural Interface Specification Language for Java". Technical Report 9806u, Iowa State University, Department of Computer Science, http://www.jmlspecs.org/.

[155] Leduc, G. and Germeau, F. (2000). "Verification of Security Protocols Using LOTOS - Method and Application". Computer Communications, 23(12), 1089-1103.

[156] Lee, D. and Yannakakis, M. (1996). "Principles and Methods of Testing Finite State Machines – A Survey". Proceedings of the IEEE, vol. 84, n. 8, August, p. 1090-1123.

[157] Lee, I., Kannan, S., Kim, M., Sokolsky, O. and Viswanathan. M. (1999). "Runtime Assurance Based on Formal Specifications". In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.

[158] Lee, P., and Necula, G. (1997). "Research on Proof-Carrying Code on Mobile-Code Security". In Proceedings of the Workshop on Foundations of Mobile Code Security, Monterey.

[159] Liebl, A. (1993). "Authentication in distributed systems: A bibliography". ACM Operating Systems Review, 27(4):31—41.

[160] Ligatti, J., Bauer, L. and Walker, D. (2005). "Edit Automata: Enforcement Mechanisms for Run-time Security Policies". International Journal of Information Security, 4(1–2).

[161] Lindholm, T. and Yellin, F. (1996). "The Java Virtual Machine specification". Web document at URL http://www.javasoft.com/docs/books/vmspec/html/VMSpecTOC.doc.html, Sun Microsystems.

[162] Longley, D. and Rigby, S. (1992). "An automatic search for security flaws in key management schemes". Computers and Security, 11(1):75—90.

[163] Lowe, G. (1997). "Casper: A compiler for the analysis of security protocols", Proceedings of The 10th Computer Security Foundations Workshop,IEEE Computer Society Press.

[164] Ludwig, H., Keller, A., Dan, A. King, R.P. and Franck, R. (2003). "Web Service Level Agreement (WSLA) Language Specification". Version 1.0, IBM Corporation (January 2003), http://www.research.ibm.com/wsla

[165] Lutz, R. (2000). "Software Engineering for Safety: A Roadmap". Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 4--11, ACM.

[166] Mahbub, K. and Spanoudakis, G. (2004). "A Framework for Requirements Monitoring of Service Based Systems". In Proceedings of the 2nd International Conference on Service Oriented Computing, NY, USA.

[167] Manna, Z., and Pnueli, A. (1992). "The Temporal Logic of Reactive and Concurrent Systems – Specification". Springer-Verlag.

[168] Manson, J., Vitek, J. and Jagannathan, S. (2005). "Dynamic Aspects for Runtime Fault Determination and Recovery". Dynamics Aspects Workshop, Chicago, USA.

[169] Mao, W. and Boyd, C. (1993). "Towards formal analysis of security protocols". Proceedings of the Computer Security Foundation Workshop VI, pages 147-158.

[170] Mascolo, C., Capra, L., Zachariadis, S. and Emmerich, W. (2002). "XMIDDLE: A Data-Sharing Middleware for Mobile Computing". In International Journal on Wireless Personal Communications, 21(1), pp.77-103. Kluwer Academic Publisher.

[171] Matheus, A. (2005). "Authorization for digital rights management in the geospatial domain". In Proceedings of the 5th ACM Workshop on Digital Rights Management (Alexandria, VA, USA, November 07 - 07, 2005). DRM '05. ACM Press, New York, NY, 55-64.

[172] McGraw, G. and Felten, E. (1999). "Securing JAVA. Getting Down to Business with Mobile Code", Chapter 3, published by John Wiley & Sons, Inc., Securing Java: Getting Down to Business with Mobile Code.

[173] McLean, J. (1994). "A general theory of composition for trace sets closed under selective interleaving functions". In Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, CA.

[174] Meadows, C. (1991). "A System for the Specification and Verification of Key Management Protocols". IEEE Symposium on Security and Privacy pages 182-197.

[175] Meadows, C. (1992). "Applying Formal Methods to the Analysis of a Key Management Protocol". Journal of Computer Security 1(1): 5-36.

[176] Meadows, C. (2003). "Formal methods for cryptographic protocol analysis: emerging issues and trends". IEEE Journal on Selected Areas in Communications, 21(1):44--54.

[177] Merritt, M. (1983). "Cryptographic Protocols". PhD thesis, Georgia Inst. of Tech.

[178] Meyer, B. (2000). "Object-Oriented Software Construction", 2nd edition. Prentice Hall, Upper Saddle River, New Jersey.

[179] Millen, J. (1995). "The Interrogator model". In Proc. 16th IEEE Symposium on Security & Privacy, pages 251—260.

[180] Millen, J. K., Clark, S. C. and Freedman, S. B. (1987). "The Interrogator: Protocol security analysis". IEEE Transactions on Software Engineering, SE-13(2):274-288.

[181] Miller, S. P., Neuman, C., Schiller, J. I. and Saltzer, J. H. (1988). "Kerberos Authentication and Authorization System". Project Athena Technical Plan, Section E.2.1, Massachusetts Institute of Technology.

[182] Mohnen, M. (2002). "A graph-free approach to data-flow analysis". In Proc. 11th CC, pages 46–61, Grenoble, France. Springer.

[183] Mok, A.K. and Liu, G. (1997). "Efficient run-time monitoring of timing constraints". In Real-Time Technology and Applications Symposium, June 1997.

[184] Moller, M., Bartetzko,D., Fischer, C. and Wehrheim, H. (2001), "Jass - java with assertions", In Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers.

[185] Moser, L. (1989). "A Logic of Knowledge and Belief for Reasoning about Computer Security". In Proceedings of the Computer Security Foundations Workshop II, pages 57--63. IEEE Computer Society Press.

[186] Moszkowski, B. (1996). "The programming language Tempura". Journal of Symbolic Computation, 22(5/6):730—733.

[187] Naldurg, P., Sen, K. and Thati, P. (2004). "A Temporal Logic Based Framework to Intrusion Detection". In Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004).

[188] Nash, M. and K. Poland (1990). “Some Conundrums Concerning Separation of Duty”. IEEE Symposium on Security and Privacy, Oakland, CA.

[189] National Computer Security Center (1985) Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28STD.

[190] National Computer Security Center (1987), “A Guide to Understanding Discretionary Access Control (DAC) in Trusted Systems”, NCSC-TG-003, Version-1.

[191] Necula, G. (1997). “Proof-Carrying Code”, In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL ’97), Paris, France.

[192] Needham, R., and Schroeder, M. (1978). “Using encryption for authentication in large networks of computers”. Communications of the ACM, 21(21):993-999.

[193] Nelson, S. and Pecheur, C. (2002) “V&V for advanced systems at NASA”, TASK NO: 10 TA-5.3.3 (WBS 1.4.4.5.3), prepared for Northrop Grumman Corp

[194] Nessett, D. M. (1990), “A critique of the Burrows, Abadi and Needham logic”. Operating Systems Review 24 (1990), 35-38.

[195] Nieh, B. B. and Tavares, S. E. (1993). “Modelling and Analyzing Cryptographic Protocols Using Petri Nets”. In: Lecture Notes in Computer Science, Vol. 718: Advances in Cryptology, AUSCRYPT'92. Springer-Verlag.

[196] Otway, D. and Rees, O. (1987). “Efficient and timely mutual authentication”. SIGOPS Oper. Syst. Rev.

[197] Pal, P., Webber, P. F., Schantz, R. E. and Loyall J. P. (2000). “Intrusion tolerant systems”. In Proceedings of the IEEE Information Survivability Workshop (ISW-2000), pages 24-26, Boston, MA.

[198] Park, J. and Chandramohan, P. (2004). “Static vs. Dynamic Recovery Models for Survivable Distributed Systems”. HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 2, IEEE Computer Society.

[199] Park, J., Sandhu, R. and Schifalacqua, J. (2000). “Security architectures for controlled digital information dissemination”. Acsac, p. 224, 16th Annual Computer Security Applications Conference (ACSAC'00).

[200] Pnueli, A. (1977). “The Temporal Logic of Programs”. In Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pages 46–77.

[201] Porras, P. A. and Neumann, P. G. (1997). “EMERALD: Event monitoring enabling responses to anomalous live disturbances”, In Proc. 20th NISTNCSC National Information Systems Security Conference, pages 353—365.

[202] Ragsdale, D., Carver, C.A., Humphries, J. and Pooch, U. (2000), “Adaptation techniques for intrusion detection and intrusion response system”. Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics at Nashville, Tennessee,pages 2344—2349.

[203] Rangan, P. V. (1998). “An axiomatic basis of trust in distributed systems”. In Symposium on Security and Privacy.

[204] Robinson, W. (2002). "Monitoring Software Requirements using Instrumented Code". In proceedings of the Hawaii Int. Conf. on Systems Sciences.

[205] Robinson, W. N. (2003). "Monitoring Web Service Requirements". In Proc. of 12th Int. Conf. on Requirements Engineering.

[206] Roscoe, A. W. (1995) "Modelling and verifying key-exchange protocols using CSP and FDR". In Proceedings of the the Eighth IEEE Computer Security Foundations Workshop (CSFW '95) (March 13 - 15, 1995). CSFW. IEEE Computer Society, Washington, DC, 98.

[207] Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.H., Jackson, D.M, and Scattergood, J.B. (1995) "Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock". TACAS 1995: 133-152

[208] Rosenblatt, B. and Dykstra, G. (2003), "Integrating content management with digital rights management - imperatives and opportunities for digital content lifecycles", White paper, Giantsteps Media Technology Strategies.

[209] Russo, A., Miller, A., Nuseibeh, B. and Kramer, J. (2002). "An Abductive Approach for Analysing Event-Based Requirements Specifications". Presented at 18th Int. Conf. on Logic Programming (ICLP), Copenhagen, Denmark.

[210] Same, D., Matt, B, Niebuhr, B., Tally, G., Whitmore, B. and Bakken, D. (2002). "Developing a Heterogeneous Intrusion Tolerant CORBA System". International Conference on Dependable Systems and Networks (DSN'02).

[211] Sandhu, R. S., Coyne, E. J., Feinstein, H. L. and Youman, C. E. (1996). "Role-Based Access Control Models". IEEE Computer vol. 29, no. 2, pp. 38-47.

[212] Sandhu, R., Ferraiolo, D. and Kuhn, R. (2000). "The NIST Model for Role-Based Access Control: Towards a Unified Standard". Proc. 5th ACM Workshop on Role-Based Access Control.

[213] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. (1997). "Eraser: A dynamic data race detector for multithreaded programs". ACM Trans. on Computer Systems, 15(4).

[214] Schlimmer, J. (editor) (2006). "Web Services Policy Framework (WS-Policy Framework)", http://www.ibm.com/developerworks/library/specification/ws-polfram/.

[215] Schneider, F. B. (1998), "Enforceable Security Policies", Cornell University Technical Report TR98-1664.

[216] Schneider, S. (1997), "Verifying authentication protocols with CSP". In Computer Security Foundations Workshop [14], pages 3--17.

[217] Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S. and DuVarney, D. (2003). "Model -Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications". ACM Symposium on Operating Systems Principles. (SOSP'03; Bolton Landing, New York).

[218] Sen, K. and Rosu, G. (2003). "Generating Optimal Monitors for Extended Regular Expressions". In Proceedings of the 3rd International Workshop on Runtime Verification (RV'03) [1], pages 162–181.

[219] Shanahan, M. (1999). "The event calculus explained". In Artificial Intelligence Today, pp.409-430, Springer.

[220] Simmons, G., and Gustavus J. (1985). "How to (Selectively) Broadcast a Secret". Proceedings of the 1985 IEEE Symposium on Security and Privacy, pp. 108-113.

[221] Sindre, G. and Opdahl, A.L. (2001) "Templates for Misuse Case Description", Proceedings of the 7 International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ'2001), Switzerland, 4-5 June 2001.

[222] Snapp, S.R., Brentano, J., Dias, G.V., Goan, T.L., Heberlein, L.T. Heberlein, Ho, C., Levitt, K.N., Mukherjee, B., Smaha, S.E., Grance, T., Teal, D.M. and Mansur, D. (1991). "DIDS (Distributed Intrusion Detection System)- Motivation, Architecture, and An Early Prototype". In Proceedings of the 14th National Computer Security Conference.

[223] Snekkenes, E. (1991). "Exploring the BAN Approach to Protocol Analysis". IEEE Symposium on Security and Privacy 1991: 171-181.

[224] Spafford, E. H. and Zamboni, D. (2000). "Intrusion detection using autonomous agents". Computer Networks, 34(4):547–570.

[225] Spivey, J.M. (1992) "The Z Notation: A Reference Manual". 2nd ed., Prentice-Hall, 1992

[226] Srinivasan, S., Dey, A., Ahamad, M. Covington, M. J., Long, W. and Abowd, G. (2001). "Securing context-aware applications using environment roles".

[227] Sun Microsystems (1999). "Java Platform Debugger Architecture Documentation", http://java.sun.com/products/jpda/doc/.

[228] Sun Microsystems (2003), "Securing Web Services - Concepts, Standards, and Requirements", White Paper.

[229] Syverson, P. (1990), "Formal Semantics for Logics of Cryptographic Protocols". In Proceedings of the Computer Security Foundations Workshop III, pages 32--41. IEEE Computer Society Press.

[230] Syverson, P. F. and van Oorschot, P. C. (1994). "On unifying some cryptographic protocols logics". In Proc. of the 13th IEEE Symp. on Security and Privacy. IEEE Comp. Society Press.

[231] Tardo, J. and Valente, L. (1996). "Mobile Agent Security and Telescript". In Proceedings of IEEE COMPCON '96, Santa Clara, California, pp. 58-63, February 1996, IEEE Computer Society Press.

[232] Tarr, P. L., Ossher, H., Harrison, W. H. and S. M. S. Jr. (1999). "N degrees of separation: Multi-dimensional separation of concerns". In International Conference on Software Engineering, pages 107–119.

[233] Tarvainen, P. (2004). "Survey of the Survivability of IT Systems". The 9th Nordic Workshop on secure IT systems, 4-5 November, Helsinki, Finland.

[234] Thane, H. (2000). "Design for deterministic monitoring of distributed real-time systems". Technical report, Malardalen Real-Time Research Centre.

[235] Thomsen, D. J. (1991). "Role-Based Application Design and Enforcement". Database Security, IV: Status and Prospects, S. Jajodia and C. E. Landwehr (eds.), North Holland, pp. 151–168.

[236] Toussaint, T. J. (1992), "Deriving the Complete Knowledge of Participants in Cryptographic Protocols". In Advances in Cryptology --- CRYPTO '91 Proceedings, pages 24--43. Springer-Verlag.

[237] U.S. Department of Commerce (1996), Federal Standard 1037, National Telecommunications and Information Administration, Institute for Telecommunications Services.

[238] van Lamsweerde, A. (1996) "Divergent Views in Goal-Driven Requirements Engineering", In proc. Viewpoints '96 – ACM SIGSOFT Workshop of Viewpoints in Software Development, October

[239] van Lamsweerde, A., Brohez, S.n, De Landtsheer, R.d and Janssens, D. (2003). "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering". Proceedings of the RE'03 Workshop on Requirements for High Assurance Systems (RHAS'03), Monterey (CA), pp. 49-56.

[240] van Lansweerde, A., Letier, E., Ponsard, C. (1997) "Leaving inconsistency". In proc. ICSEC'97 workshop on "Living with Inconsistency", May 17

[241] van Oorschot, P. (1993). "Extending cryptographic logics of belief to key agreement protocols". Proc. 1 st ACM Conference on Communications and Computer Security (Fairfax, Virginia, Nov. 3-5).

[242] Ventuneac, M., Coffey, T. and Salomie, I. (2003). "A policy-based security framework for Web-enabled applications". In Proceedings of the 1st international Symposium on information and Communication Technologies, ACM International Conference Proceeding Series, vol. 49. Trinity College Dublin, 487-492.

[243] VeriSign (2005). "VeriSign Code Signing for Netscape Object Signing, in Business Guide", Chapters 2,3, VeriSign, http://www.verisign.com/static/030997.pdf.

[244] Verissimo, P. E., Neves, N. F. and Correia, M. P. (2003). "Intrusion-tolerant architectures: Concepts and design". In R. Lemos, C. Gacek, and A. Romanovsky, editors, Architecting Dependable Systems, volume 2677 of Lecture Notes in Computer Science, pages 3--36. Springer-Verlag.

[245] Visser, W., Havelund, K., Brat, G. and Park, S. J. (2000). "Model Checking Programs". In Proceedings of ASE-2000: The 15th IEEE Conference on Automated Software Engineering. IEEE CS Press. Grenoble, France.

[246] Wagelaar, D. (2004). "Towards a context-driven development framework for ambient intelligence". In Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (withICDCS'04), Tokyo, Japan.

[247] Wedel, G. and Kessler, V. (1996). "Formal Semantics for Authentication Logics". In: E. Bertino, H. Kurth, G. Martello, and E. Montolivo (eds.): Proc. ESORICS'96. pp. 219--241, LNCS 1146.

[248] Weimer, W, and Necula, G.C. (2004). "Finding and preventing run-time error handling mistakes". In 19th Anual ACM Conference on Object-Oriented programming, Systems, Languages, and Applications (OOPSLA '04), pages 419-431

[249] Wilikens, M., Feriti,S., Sanna, A. and Masera, M. (2002). "A context-related authorization and access control method based on RBAC: A case study from the health care domain". In Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT).

[250] Wing, J. (1998). "A Symbiotic Relationship Between Formal Methods and Security". Technical Report, CMU-CS-98-188.

[251] XrML 2.0 Technical Overview (2002), Version 1.0

[252] Yahalom, R., Klein, B. and Beth, T. (1993). "Trust relationships in secure systems - A distributed authentication perspective". In Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy, pages 150—164.

[253] Yang, Z., Cheng, B. H., Stirewalt, R. E., Sowell, J., Sadjadi, S. M. and McKinley P. K. (2002). "An aspect-oriented approach to dynamic adaptation". In Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02).

[254] Yao, A.C. (1982). "Theory and application of trapdoor functions". In Proc. of the 23th Annual IEEE Symposium on Foundations of Computer Science, pp. 80—91.

[255] Yellin, F. (1996). "Low-level security in Java". Web document at URL http://www.javasoft.com/sfaq/verifier.html, Sun Microsystems.

[256] Zakinthinos, A. and Lee, E. S. (1997). "A general theory of security properties".. In Proceedings of the 1997 IEEE Symposium on Security and Privacy SP. IEEE Computer Society, Washington, DC, 94.

[257] Zhou, J. and Gollmann, D. (1996). "A fair non-repudiation protocol". In Proceedings of the IEEE Symposium on Research in Security and Privacy [IEE96], pages 55-61.

[258] Aickelin U., Cayzer S., (2002). "The Danger Theory and Its Application to Artificial Immune Systems". In Proc. First Int. Conf. On Artificial Immune Systems, Canterbury

[259] Aickelin U., Greensmith J., Twycross J., (2004). "Immune System Approaches to Intrusion Detection – A Review", in Proc. Third Int. Conf. On Artificial Immune Systems (ICARIS),

[260] Balthrop, J. Forrest, S. and Glickman, M., (2002). "Revisiting LISYS: Parameters and Normal Behavior.". Proceedings of the 2002 Congress on Evolutionary Computation.

[261] Bettini, L. and De Nicola, R., (2002) "A Middleware for Secure Distributed Tuple Spaces"

[262] Bravetti, M., Busi, N, Gorrieri, R, Lucchi, R., and Zavattaro, G., (2004) "Security Issues in the Tuple-Space Coordination Model", In Proc. of the second International Workshop on Formal Aspects in Security and Trust (FAST'04) Kluwer Academic Press

[263] Bryce, C. and Cremonini, M., (2001). "Coordination and Security on the Internet", in Coordination of Internet Agents, A.Omicini, F.Zambonelli, M.Klusch, R.Tolksdorf (Eds.), Springer

[264] Bryce, C., Oriol, M., and Vitek, J., (1999) "A Coordination Model for Agents based on Secure Spaces".

[265] Cabri, G., Leonardi, L., and Zambonelli, F., (1998). "Reactive Tuple Spaces for Mobile Agent Coordination", Lecture Notes in Computer Scinece,

[266] Conham R.O., Tyrrell A.M., (2002) "A Multilayered Immune System for Hardware Fault Tolerance within an Embryonic Array", In Proc. First Int. Conf. On Artificial Immune Systems, Canterbury

[267] Dasgupta, D. and Brian, H., (2001). "Mobile Security Agents for Network Traffic Analysis". Published by the IEEE Computer Society Press in the proceedings of DARPA Information Survivability Conference and Exposition II (DISCEX-II), Volume: 2, Page(s): 332-340, Anaheim, California, June 12-14

[268] Dasgupta, D., Gonzalez, F., Yallapu, K. and Kaniganti, M., (2003). “Multilevel Monitoring and Detection Systems (MMDS)”. in the proceedings of the 15th Annual Computer Security Incident Handling Conference (FIRST), Ottawa, Canada June 22-27

[269] de Castro, L.N. and Timmis, J., (2002) “Artificial Immune Systems: A New Compuatational Intelligence Approach”. Springer

[270] Esponda F., Forrest S., and Helman P., (2004). “A Formal Framework for Positive and Negative Detection Schemes”, IEEE Transaction on Systems, Man, and Cybernetics, Vol.34, No.1

[271] Forrest, S., Perelson, A.S., Allen, L., and Cherukuri, (1994) "Self-Nonself Discrimination in a Computer." In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA: IEEE Computer Society Press

[272] Forrest, S., Hofmeyr, S.A., Somayaji, A., and Longstaff, T.A., (1996). "A Sense of Self for Unix Processes.", In Proceedings of 1996 IEEE Symposium on Computer Security and Privacy.

[273] 'haeseleer, P. D. Forrest, S. and Helman, P. (1997) “A Distributed Approach to Anomaly Detection."

[274] Forrest, S., Somayaji, A., and Ackley, D., (1997). "Building Diverse Computer Systems." In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (1997).

[275] Forrest, S., Hofmeyr, S.A., Somayaji, S, (1997) “Computer Immunology”, Communication of ACM, Vol.40 No.10, October 1997.

[276] Gonzalez, F.A., and Dasgupta D., (2003). “Anomaly detection Using Real-Valued Negative Selection”, In Special Issue of the Journal of Genetic Programming and Evolvable Machines, Vol.4 No.4

[277] Gonzalez, F.A., Dasgupta D., and Kozma R., (2002). “Combining Negative Selection and Classification Techniques for Anomaly Detection”, in Journal IEEE Transactions on Evolutionary Computation Vol.1

[278] Hofmeyr, S. and Forrest, S., (2000). "Architecture for an Artificial Immune System." Evolutionary Computation 7(1), Morgan-Kaufmann, San Francisco, CA, pp. 1289-1296 (2000).

[279] Hofmeyr, S., Forrest, S., and Somayaji, A. (1998). "Intrusion Detection Using Sequences of System Calls." Journal of Computer Security Vol. 6, pp. 151-180

[280] JavaSpaces, “JavaSpaces Service Specification”, (2005) (available at http://www.jini.org/nonav/standards/porter/doc/specs/html/js-spec.html)

[281] Kephart, J.O., Sorkin G.B., Swimmer M., White S.R., (1997). “Blueprint for a Computer Immune System”, Virus Bullettin International Conference, San Francisco

[282] Kim J., Wilson W.O., Aickelin U., McLeod J., (2005) “Cooperative Automated worm Response and Detection ImmuNe Algorithm (CARDINAL) inspired by T-cell Immunity and Tolerance”, in Proc. of the fourth Int. Conf. On Immune Systems (ICARIS)

[283] Le Boudec J.Y., Sarafijanovic S., (2003) “An Artificial Immune System Approach to Misbehaviour Detection in Mobile Ad-Hoc Networks”, Tech.Rep. IC/2003/59

[284] Menezes, R., and Wood, A., (1999). “Garbage collection in Linda Using Tuple Monitoring and Process Registration”

[285] Nunes de Castro, L., and Von Zuben, F.J., (1999). “Artificial Immune Systems: Basic Theory and Applications”, Technical Report TR-DCA 01/99

[286] Nunes de Castro, L., and Von Zuben, F.J., (2000). “Artificial Immune Systems: Survey of Applications”, Technical Report DCA-RT 02/00

[287] Omicidi, A., and Zambonelli, F., (1998). “Tuple Centre for the Coordination of Internet Agents”, ACM

[288] Warrender, C., Forrest, S., Pearlmutter., B. (1999). “Detecting intrusions using system calls: Alternative data models” 1999 IEEE Symposium on security and Privacy.

[289] Meadows, C., (1994). “Formal Verification of Cryptographic Protocols: A Survey”. Proc. 4th International Conference on the Theory and Applications of Cryptology: Advances in Cryptology, 135-150.

[290] Roscoe, A.W., and Goldsmith, M.H., (1997). “The perfect spy for model-checking crypto-protocols”. Proceedings of DIMACS workshop on the design and formal verification of crypto-protocols.

[291] Dill, D.L., Drexler, A.J. Hu, A.J. and Yang, C.H., (1992). "Protocol Verification as a Hardware Design Aid". IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525.

[292] Mitchell, J.C., Mitchell, M. and Stern, U., (1997). "Automated analysis of cryptographic protocols using Murphi," IEEE Symposium on Security and Privacy.

[293] Tatebayashi, M., Matsuzaki, N., and Neuman, D.B. Jr., (1989). “Key Distribution Protocol for Digital Mobile Communication Systems”. In G. Brassard, editor, Proceedings of Advances in Cryptography | CRYPTO'89, volume 435 of Lecture Notes in Computer Science, pages 324-334. Springer-Verlag.

[294] Kohl, J and Neuman. B.C., (1993). “The Kerberos Network Authentication Service (Version 5)”. Internet Request for Comments RFC-1510. September.

[295] Ghezzi, C., and Kemmerer, R. (1991). “ASTRAL: An assertion language for specifying real-time systems”. Proceedings of the 3rd European Software Engineering Conference, pp.122-146.

[296] Dang, Z., and Kemmerer, R.A. (1997) “Using the ASTRAL model checker for cryptographic protocol analysis”. In Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols, September.

[297] Dang, Z. and Kemmerer, R.A., (1999). "Using the ASTRAL model checker to analyze Mobile IP," Proc. of ICSE'99, pp. 132-141.

[298] Dai, Z., He, X., Ding, J., and Gao, S., (2004). “Modeling And Analyzing Security Protocols In Sam: A Case Study”. Proc. of the 8th IASTED International Conference on Software Engineering and Applications, Cambridge, MA, USA, November 9-11.

[299] Heitmeyer, C., Kirby, J., Labaw, B., and Bharadwaj, R., (1998). “SCR*: A toolset for specifying and analyzing software requirements”. In Proc. Computer-Aided Verification, 10th Annual Conf. (CAV’98), Vanvouver Canada, 1998

[300] Archer, M., Heitmeyer, C., and Riccobene, E. (2000). “Using TAME to prove invariants of automata models”. Case studies. In Proc. 2000 ACM SIGSOFT Workshop on Formal Methods in Software Practise (FMSP’00), August

[301] Bharadwaj, R., and Sims, S. (2000). “Salsa: Combining constraints solvers with BDDs for automatic invariant checking”. In Proc. Tools and Algorithms for the construction and Analysis of Systems (TACAS’ 2000), Berlin, March.

[302] Heitmeyer, C.L. (2001). “Applying Practical Formal Methods to the Specification and Analysis of Security Properties”. MMM-ACNS 2001: 84-89

[303] Gargantini, A., and Heitmeyer, C.L., (1999). “Using Model Checking to Generate Tests from Requirements Specifications”. ESEC / SIGSOFT FSE 1999: 146-162

[304] Abrial J-R and Mussat L, (1998). “Introducing Dynamic Constraints in B”. D.Bert (Editor): Proceedings of the Second International B Conference B'98: Recent Advances in the Development and Use of the B Method, Springer.

[305] Lanet, J.P (1998). “Using the B Method to Model Protocols”. In Proc. AFADL 98, pages 79-90

[306] Motre, S., and Teri, C. (2000). “Using B Method to Formalize the Java Card Runtime Security Policy for a Common Criteria Evaluation”. In Proc. of 23rd National Information Systems Security Conference (NISSC 2000), Baltimore, USA, October 16-19.

[307] Ahrendt, W., Baar, T., Beckert, B., Giese, M., Habermalz, E., Hhnle, R., Menzel, W., and Schmitt, P.H. (2000). “The Key approach: integrating design and formal verication of Java Card programs”. In Proceedings of the Java Card Workshop, co-located with the Java Card Forum, Cannes, France.

[308] Girard, P. and Lanet, J.-L. (1999). “New Security Issues raised by Open Cards”. In Information Security Technical Report, Vol4, N2, pp.: 19-27

[309] Guilley, S., and Pacalet, R. (2004). “SoC Security: a War against Side-Channels”. Annals of the Telecommunications, July/August 2004.

[310] Witteman, M. (2002). “Advances in smart card security”. Information Security Bulletin, pg 11-22

[311] Hall, A., and Chapman, R., (2002). “Correctness by Construction: Developing a Commercial Secure System”. IEEE Software 19(1): 18-25.

[312] Landwehr, C., Bull, A., McDermott, J., and Choi, W. (1994). “A taxonomy of computer program security flaws, with examples”. ACM Computing Surveys, 26(3):211--255, 1994.

[313] Bicarragui, J., Dick, J., and Woods, E. (1996). “Quantitative analysis of an application of formal methods”. In FME ’96: Industrial Benefit and Advances in Formal Methods, volume 1051 of Lecture Notes in Computer Science, pages 60-73, Springer-Verlag

[314] Nicola and van Spanje (1990). “Comparative analysis of different models of checkpointing and recovery”. IEEE Transactions on Software Engineering, 16:807-821, August 1990

[315] Ziv, A., and Bruck, J. (1997). “An On-Line Algorithm for Checkpoint Placement”. IEEE Transactions on Computers, vol. 46(9), pp. 976--985, Sept.

[316] Avizienis, A., Laprie, J.C., Randell, B. and Landwehr C. (2004). “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Transaction on dependable and secure computing, Vol.1, No.1, January-March 2004

[317] W. Torres-Pomales, (2000) “Software Fault Tolerance: A Tutorial”. Technical Report TM-2000-210616, NASA, October

[318] Schneier, B. (1998). “Security Pitfalls in Cryptography”. Essay, http://www.schneier.com/essay-028.html

[319] Ammann, P., Jajodia, S. and I. Ray, (1997). “Applying Formal Methods to Semantic-Based Decomposition of Transactions”. ACM TODS, Vol. 22, No. 2, June 1997.

[320] Minsky, N. H., Minsky, Y. M., Ugurenau, V. (2000). “Making Tuple Spaces Safe for Heterogeneous Distributed Systems”. Proc. SAC’2000, Como, Italy.

[321] Gelernter, D. (1985). “Generative communication in Linda”. ACM Transactions on Programming, 2(1):80—112.

[322] Kim, J. and Bentley, P. J. (2001). “Evaluating Negative Selection in an Artificial Immune System for Network Intrusion Detection”. Genetic and Evolutionary Computation Conference 2001 (GECCO-2001), San Francisco, pp.1330 - 1337, July 7-11.

[323] Matzinger, P. “The Real Function Of The Immune System”, available from http://cmmg.biosci.wayne.edu/asg/polly.html.

[324] Picco, G. P., Murphy, A. L. and Roman, G.-C. (1999). “Lime: Linda Meets Mobility”. In D. Garlan, editor, Proceedings of the 21 st International Conference on Software Engineering.

[325] Sathyanath, S. and Sahin, F. (2002). AISIMAM - An AIS based Intelligent Multi Agent Model and Its Application to Mine Detection Problem, Proceedings of the ICARIS 2002 1st International Conference on Artificial Immune Systems, Canterbury, UK, September 9 – 11.

[326] Williams, P. D., Anchor, K. P., Bebo, J. L., Gunsch, G. H., Lamont, G. B. (2001). “CDIS: Towards a Computer Immune System for Detecting Network Intrusions”. Recent Advances in Intrusion Detection 2001: 117-133.