ARCHITECTURE-CENTRIC TESTING FOR SECURITY

by

SARAH AL-AZZANI

A thesis submitted to
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
December 2013

Abstract

This thesis presents a novel architecture-centric approach, which uses Implied Scenarios (IS) to detect design-vulnerabilities in the software architecture. It reviews security testing approaches, and draws on their limitations in addressing unpredictable behaviour in the face of evolution. The thesis introduces the concept of Security IS as unanticipated (possibly malicious) behaviours that indicate potential insecurities in the architecture. The IS approach uses the architecture as the appropriate level of abstraction to tackle the complexity of testing. It provides potential for scalability to test large scale complex applications. It proposes a three-phased method for security testing: (1) Detecting design-level vulnerabilities in the architecture in an incremental manner. This is done via investigating emergent behaviours (i.e. ISs) in the composition of functionalities as functionalities evolve. (2) Classifying the impact of detected ISs on the security of the architecture. (3) Using the detected ISs and their impact to guide the refinement of the architecture. The refinement is test-driven and incremental, where refinements are tested before they are committed. The thesis also presents SecArch, an extension to the IS approach to enhance its search-space to detect hidden race conditions. It is concerned with predicting further valid conditions in the face of real parallelism in distributed systems with respect to non-FIFO queues.

The thesis reports on the applications of the proposed approach and its extension to three case studies for testing the security of distributed and cloud architectures in the presence of uncertainty in the operating environment, unpredictability of interaction and possible security IS. The applications demonstrate novelty in the way security testing addresses emergent behaviour in applications which are characterised with dynamism, heterogeneity, openness, scale and unpredictability in operation and their evolution trends. We have drawn on these case studies to evaluate the thesis.

Acknowledgement

I owe my deepest gratitude to my supervisor, Dr. Rami Bahsoon, for his endless support and guidance. His enthusiastic encouragement, useful critiques, and willingness to give time so generously have been very much appreciated. Without his consistent support this thesis would not have materialised. I am grateful to my RSMG member Prof. Xin Yao for his constructive comments and insightful suggestions.

I wish to thank my parents for their support and encouragement throughout my study. Special thanks to my entertaining friends Hana, Esra, Khuloud, Annie, Shen and her baby Imad, Mahamat, Siti, Guru, Lenka, Funmi & Funmi, Ben, Jeff, Minlue and many more that have made my journey very enjoyable.

Contents

1

Introduction

1.1 Background

The ability to deliver reliable software systems of high quality within budget and schedule continues to challenge most IT organisations. Developing and maintaining complex software solutions in a distributed environment imposes challenges in achieving secure software systems, whether it is due to concurrency and lack of synchronisation, varying degrees of component trust or distribution of data. With the current growth of software, applications are becoming too bloated to be tested effectively; true software security is achievable only when all known aspects of the software are understood and verified to be predictably correct. This requires verifying the correctness of the software behaviour under a wide variety of conditions, including hostile conditions (eg, near trust boundaries). Many security failures occur in stressed environments, but are often neglected during testing because of the difficulty in simulating these conditions in real time [178]. In addition, because testing is often performed under immense time pressure, minimal time is devoted towards test-case crafting to maximise exploration of program state space and intelligently drive a program into potential vulnerable states. Attempting to perform testing in an ad-hoc manner does not address the fact that some components tend to be more problematic than others; and thus the testing must be performed systematically.

Lack of security expertise amongst software engineers results in a software design with minimal security consideration [113]. Software developers are generally not provided with the adequate information on how to develop secure applications [191, 27]. A metric collected from Microsoft Developer Research [64] stated that: '64% of developers are not confident in their ability to write secure applications'. This emphasises the problem of performing adequate security testing, because security testing is fundamentally different to traditional functional testing. In 2011, Warren Axelrod [20] highlighted the significant difference between functional testing for security and functional security testing (FST), where he stated that functional testing for security 'attempts to ensure that the functionality of security matches requirements', and FST 'which is designed to ferret out the malfunctioning of applications that might lead to security compromises'. He emphasised

the lack of research in the FST due to the skills, effort and cost it requires, and that the importance of negative functional testing is not generally recognised by business management and IT development managers. Security testing aims to uncover unspecified/hidden behaviour within the system, whereas functional testing certifies whether or not the system behaves as intended. This implies that, regardless of the time and cost put into security testing, the discovery of vulnerabilities cannot be guaranteed.

Software vulnerabilities jeopardize intellectual property, consumer trust and business operations and services. The integrity of key assets depends upon the reliability and security of the software that controls those assets. Vulnerabilities are defects/weaknesses in the system that may arise at varying levels (eg, design-level, operational-level, implementation-level), where different testing methodologies address different sets of vulnerabilities. For example, static-analysis tools search for implementation-level vulnerabilities, and penetration testing searches for operation-level vulnerabilities. Even though these methodologies proved successful at uncovering their targeted vulnerabilities, it was observed that there is too much focus on detecting implementation-level vulnerabilities [170, 194], such as buffer-overruns and invalid-input, whereas very little attention is paid to design vulnerabilities (often referred to as flaws such as transitive-trust mistakes) [194], although design vulnerabilities cannot be addressed at later stages in the development cycle, unless the system is redesigned [190]. These design vulnerabilities tend to have much greater impact in terms of exploits and security consequences [12].

According to Hoglund et al. [76], many attacks rely on exploiting design vulnerabilities. This belief is backed up by statistical evidence that indicates the importance of creating secure software designs, as opposed to adding security features after the design is built. For example, the SANS institute [82] reported that the number of vulnerabilities being discovered in applications is far greater than the number of vulnerabilities discovered in operating systems, totalling approximately 77% in some estimations [52]. Statistical data published by the National Institute of Standards and Technology (NIST) [132] indicate that the number of vulnerabilities in software applications has doubled in

Table 1.1: NIST statistics: Number of application-layer vulnerabilities reported per year

Year	No. of Software Vulnerabilities
2006	6,608
2007	6,514
2008	5,632
2009	5,732
2010	4,639
2011	4,150
2012	5,289
Average	38564/7 = 5509 per year

Year	No. of Software Vulnerabilities
2000	1,020
2001	1,677
2002	2,156
2003	1,527
2004	2,451
2005	4,931
Average	13762/6 = 2293 per year

the past 7 years, with an average of 5509, as compared to the initial 5 years of the 2000s, which averaged 2293 vulnerabilities. Table 1.1 presents the number of application layer vulnerabilities reported between 2000 and 2012. It is believed that one reason behind these numbers is that designers often treat security as an add-on feature [163], rather than as a fundamental aspect of software engineering. Others [76] attribute the increase to the growing connectivity between networks and applications, making it easier for attackers to reach the application remotely. On the other hand, Cenzic [32] saw no significant decrease in 2011; they believe that hackers continue to find new and sophisticated ways to break application security at a steady rate. While secure coding practices have not improved the ability to keep vulnerability rates steady, these statistics raise the bar for emergent attention to security software testing at the design level.

The question that needs to be raised is: Why do application security problems exist? According to Martin Borrett from IBM [27], IT security professionals mostly come from network and infrastructure backgrounds, with very little experience in application development; their networking security solutions (such as network scanners and firewalls) cannot block application attacks or detect application vulnerabilities. The conclusion according to Dunphy et al. [46] is unavoidable: 'any notion that security is a matter of simply protecting the network perimeter is hopelessly out of date'. Much of the work in the area of security has come from the cryptography community [163], while other areas of computer science (such as computer networks and theory) have also contributed to the solution of security problems; however, software engineering is believed to have made

a smaller contribution [163]. Although a variety of security approaches have arisen in the past decade within software engineering, research is necessary in regards to testing functionality for hidden behaviours [20].

To address design vulnerabilities, and to draw security testing towards investigating functional-misbehaviour, it is necessary to work at the architecture and design level. Architecture-centric testing for security is the process of establishing confidence in the security and dependability of the architecture. Its aim is to test the extent to which the architecture posture (ie, structure, behaviour and composition, etc) is resilient to likely attacks, and to ensure that it behaves as intended. The process involves: (1) detecting malicious behaviours; and (2) using these malicious behaviours to guide the architecture refinement process. The analogue between code-level testing and architecture-level testing is rather similar: Instead of working at the code level, and debugging an application and then fixing the vulnerable code, we work at the architecture level, debugging it by searching for malicious behaviours, and then fixing it through a refinement process. Security experts consider the architecture and design to be the single most critical phase of the secure design life cycle [12, 163], because good decisions made during this phase yield an approach and structure that are resilient and resistant to attacks. The architecture provides a sufficient level of abstraction (such as message passing and interfaces) that hides away the unnecessary details of complex code-level interactions. However, the challenge in securing an architecture is that the architecture must not only address security issues but it must also be flexible and resilient under constantly changing security conditions. For example, the longer duration of data preservation means some of the security codes and policies may change, and security requirements will have to evolve over time as user's requirements change. Thus, it is necessary to incrementally incorporate security testing in the refinement processes of the architecture and design, while taking into consideration the incomplete knowledge about the architecture in the early stages of development. As architecture level security testing aims to reveal defects and threats at the architecture level, this can be a key to driving the architecture refinement to build a secure architecture.

1.2 Problem Scope

Because the scope of problems associated with security testing is large, this thesis is concerned with the following problems:

1. As discussed in the previous section, one of the leading problems in security testing is that security testing is often not integrated in the design phase of the system development; as a result, fixing design vulnerabilities can be costly, leading to the common choice of 'find and patch' in security testing. Patching is a process of hiding symptoms of the problem, as opposed to fixing its root causes and correcting the design. In many situations, patching does not offer a solution to design vulnerabilities, and it is also difficult in practice to enumerate all symptoms in order to prevent the vulnerability from being exploited. This emphasises the need to bring security testing into the early design stages of development so that design vulnerabilities can be addressed before the system is built. There is also the need for a proactive approach to reveal unknown and unspecified behaviour within a design. Currently, the norm is to react to threats, and to use pre-known threats to exercise the system or test its functionality for conformance. This leaves security testers always one step behind the attackers, where we only fix what the attacker has already exploited. A proactive approach allows us to build security into the design by, for example, exploiting implicit assumptions made by designers about possible component interactions. Given a set of specifications, we need to verify that all relevant components collaborate correctly to ensure that the global security of the system is achieved. When we put small components together, predicting the consequences of their composition can be difficult. Thus, we need to have ways that let us model, predict and evaluate the effects of component composition on the system's security as the system evolves.

2. Security testing is fundamentally different from traditional testing because it emphasises what an application should not do rather than what it should do. We therefore

distinguish between *positive requirements* such as 'disabling user accounts after three unsuccessful login attempts' and *negative requirements* such as 'the system should prevent unauthorised users from accessing system resources' [56]. Unlike functional testing, security testing places greater emphasis on negative requirements. To apply the standard testing approach to negative requirements, it would be necessary to create every possible set of negative conditions that tests the requirement, which is infeasible because it is not practical to reliably enumerate all such possibilities. This is also challenging because it is not always possible to map a requirement to a specific software artefact when the requirement is not implemented in a specific place.

As a result, we are interested in a proactive architecture-level approach to allow us to search for hidden behaviours in the architecture, so that we can anticipate their presence before they are exploited. We need to discover what the application can do beyond the specified behaviour, because security is about testing for hidden interactions as opposed to known specified behaviour. In short, the approach can reveal the following benefits:

- shall support threat detection on the architecture, to allow for a proactive approach for detecting design vulnerabilities caused by unpredictable functional compositional behaviour;
- shall guide the refinement of the architecture incrementally; this allows us to build security into the design, and to make informed decisions about changes in the architecture;
- shall allow for iterative development to take into account the nature of software development and the lack of complete knowledge during the early stages of development;
- shall raise the level of abstraction of security testing as a way to render potential scalability.

1.3 Towards Architecture-centric Security Testing

The design of software for concurrent systems is generally complex, with a high possibility that subtle errors will cause erroneous behaviour [35]. Some of these errors may have catastrophic security consequences in terms of money, time or even human life. For example, design/architectural mismatches are potentially exploitable weaknesses. Often these mismatches occur between a component and its operating environment; for example, because the component makes assumptions beyond what the environment can provide. Attackers intentionally probe for unspecified behaviours in the system [177]. They attempt to make the application behave unexpectedly, and then determine the attacks associated with that behaviour.

The challenge in achieving compositional security is that security is a global property; yet the only way we know how to build big systems is by using smaller pieces. When we put small pieces together, predicting the consequences of their composition is difficult [194]. Many vulnerabilities are believed to arise from unexpected interactions between different system components [144]; attackers *combine several legitimate behaviours to produce emergent abusive behaviour* [194]. In the real world, a designer might explicitly specify that an application performs scenarios *X*, *Y* and *Z*. As these scenarios are composed together, a fourth scenario, *W*, might arise as a consequence of their composition. Such an unspecified scenario, namely W, might result in a security failure if exploited by an attacker. These undocumented scenarios are known as '*implied scenarios*' (ISs) [14], and may lead to design vulnerabilities. *ISs indicate gaps in the specification when the behaviour of a system is specified from a global perspective, yet it is expected to be implemented component-wise, with a local view of the system.* If the specified architecture does not provide components with a rich enough local view of what is happening at a system level, components may not be able to enforce the intended system behaviour. Effectively, each component, from its local perspective, may believe that it is behaving correctly, yet from a system perspective the behaviour may not be desirable.

The thesis argues that architecture-centric testing for security should involve testing for vulnerabilities that arise from unexpected interactions between components. In this context, testing for security should test the system for behaviours not explicitly specified in the system model. It advocates a test-driven approach for security testing using IS detection, and benefits from its ability to detect hidden assumptions and possible interleavings in *incomplete* system models. The term *incomplete* system models refers to incomplete system specifications. This often occurs because a complete knowledge about the system may not be present at the initial stages of development, or when the system evolves or operates in dynamic and open environments. When the system evolves, changes such as adding or modifying a functionality, may introduce new behaviours that violate the existing security of the architecture. The thesis supports incremental refinement of the architecture upon changes made to the system functionality. This assists the impact of functional requirements on the security of the architecture.

The thesis introduces the application of IS approach [5] for security testing to reveal unexpected interactions between system components. Drawing on case studies from different application domains (reported in [6, 7, 155]), it shows the effectiveness of the IS approach for security testing, to guide the refinement process of the architecture with security in mind, and to inform the selection of more secure architectures. It explores the concept and the foundation underlying IS, how and why they arise, and their role in security testing. There is a general lack of research in analysing dependability with respect to interactions across multiple views [60], the thesis exploits the connections between system behavioural and structural models to provide extended mean to analyse and understand the security of concurrent systems [7]. Each of these models focuses on certain aspects of interactions; thus, integrating information gathered from multiple views (ie, multiple levels of abstraction and from a variety of perspectives) provides adequate multi-dimensional representation of the pre-developed system that features reduced design vulnerabilities. Incorporating multiple-views for security analysis allows for understanding the possible consequences of negative behaviour across other views.

It is necessary to complement security testing with automation support, because testing is a phase during which budget and schedules are tight. Full automation of security testing has failed thus far, except in a few cases [183]. The main reason is that hacking is a creative process, and creativity cannot be easily automated. In addition, each application has its own design, and each design has its own weaknesses and must therefore be tested individually. Even though some security experts believe that architecture-level flaws can currently be found only through human analysis [12], the nature of ISs is generally difficult to detect manually. Thus, we show how we can benefit from semi-automation to reduce the time needed to search for these scenarios. Unlike fully human-centric approaches, automation is less prone to errors and the overlooking of threats.

1.3.1 Research Objectives

The goal of this thesis is to develop an architecture-centric approach that systematically tests the security of software architecture in an incremental fashion. Incremental fashion means adapting to continuous changes that maybe imposed on the architecture due to changes in the functional requirement or refinements. Taking a test-driven approach for security testing, the intention is to proactively assess the impact of incremental functional changes on the security posture of the architecture. The thesis aims to understand the ability of the approach to: (1) detect design vulnerabilities on the architecture, (2) guide the refinement process of the architecture, (3) assess the impact of the incremental changes on the overall security, (4) inform the architecture selection of more secure architectures, (5) be used in iterative agile-like developments and to (5) provide assistance for testers to reveal complex design vulnerabilities that may be hard to detect manually.

The thesis aims to complement existing security testing approaches such as code-level testing, or network-level testing, and to enhance the process by aiding testers to identify security vulnerabilities using the system's architecture and design. The research questions addressed in this thesis are:

- Can the IS approach detect design vulnerabilities in the architecture?
- Can the IS approach provide guided systematic refinement process to build security into the architecture?
- Can the IS approach provide early feedback on the security of decisions made in the architecture?
- Can the search-space of IS approach be enhanced to incorporate multiple-views for security analysis?

1.3.2 Thesis Contribution

The thesis explores the field of security testing. It probes for understanding the state-of-art and -practice in testing software systems for security. It reviews approaches, which practitioners and researchers have considered in security testing. The review has established the evidence that the route for testing security software has been lacking systematic guidance and is performed on low-level software artefact such as design and code. Moreover, existing approaches have been found to be limited in their scalability and adaptivity to changes in functional requirements. Their use of predefined specifications and test cases renders the meaningfulness and effectiveness of security testing 'myopic'. This is because: (i) the landscape of security tends to evolve with changes in the threat landscape; (ii) many emerging behaviours, which can threaten the security posture of the systems, occur at runtime and are therefore difficult to anticipate at design time; (iii) while code and low-level design artefacts tend to be effective for testing programming bugs and assembling components into security subsystems, many of the vulnerabilities are architectural in nature [76]. They are anticipated to be 50% of the total number of vulnerabilities [114].

The thesis is the first effort towards an architecture-centric testing for security. It pursues the architecture as an artefact for security testing as such is believed to (i) offer the appropriate level of abstraction to tackle the complexity of testing, (ii) to reason about secure composition, interaction, concurrency and emerging behaviour and (iii) to provide

the potential for scalability (ie, facilitating the testing for security in large-scale complex applications). The thesis explores the requirements for security testing. It proposes a novel solution, which leverages on the work of Implied Scenarios (IS) [185]. The thesis introduces the concept of Security IS. Security ISs are IS with unanticipated (possibly malicious) behaviours that indicate potential insecurities in the architecture. It proposes a three-phased method for security testing: (1) Detecting design-level vulnerabilities in the architecture in an incremental manner. This is done via investigating emergent behaviours (ie, ISs) in the composition of functionalities as functionalities evolve. (2) Classifying the impact of detected ISs on the security of the architecture. The steps required to perform a systematic classification of threats are outlined. (3) Using the detected ISs and their impact to guide the refinement of the architecture. The refinement process is test-driven and incremental, where refinement cycles are tested before they are committed. The method provides a proactive and early feedback on the security of the changes applied to the architecture, such that testers are able to make informed decisions about the refinements.

To enhance the search-space for threats, the thesis presents an extension to the IS approach to detect hidden race conditions, allowing for testing for security with the presence of negative behaviour. The extension, called SecArch, is concerned with predicting more valid conditions in the face of real parallelism in distributed systems with respect to non-FIFO queues. This is done by moving from purely dynamic behaviour models (LTS models) to structural MSC models to preserve structural properties that are used to detect race conditions, and generate further ISs.

This thesis reports on the applications of the proposed approach and its extension to three case studies for testing the security of distributed and cloud architectures in the presence of uncertainty in the operating environment, the unpredictability of interactions and possible security ISs. The applications demonstrate novelty in the way in which security testing is used to address emergent behaviour in applications characterised by dynamism, heterogeneity, openness, scale and unpredictability in their operational and

evolutionary trends. We have drawn on these case studies to evaluate the thesis. We explored the fitness of the approach for detecting threats in the architecture and guiding the architecture refinement process. We experimented with the approach's ability to inform the selection of a more secure architecture, and we demonstrated that security testing can be performed even when we do not have complete knowledge about the behaviour of the components. We verified the claim that ISs are behaviours that can result in security implications for the architecture, and that testing for these ISs is critical in order to predict the compositional security of the dynamic behaviour of the architecture. We demonstrated that the detection of ISs supports a proactive application of security testing, in which we detect design vulnerabilities based on the architecture itself, without relying on pre-known threats already explored by attackers. We have verified that security testing can be performed in iterative development cycles; this allowed us to build security into the design by making informed decisions about the impact of changes made to the architecture. We have also reflected on the potential of the approach with respect to criteria like ease of learning, effectiveness, scalability and applicability to further evaluate the thesis.

1.4 Thesis Publications

The work presented in this thesis is based on and extends several papers that have been published in the last three years in premier venues in software architecture, and security software engineering. This thesis should be regarded as the definitive account of the work.

- Conference

1. S.Al-Azzani and R.Bahsoon, *Semi-automated detection of architectural threats for security testing*, in Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium (ACM 2009)

2. S.Al-Azzani and R.Bahsoon, *Using implied scenarios in security testing*, in Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, SESS

2010, Cape Town, South Africa, (ACM 2010) [Selection rate: 31%]

3. S.Al-Azzani, R.Bahsoon *SecArch*: *Architecture-level Evaluation and Testing for Security*, 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2012, Helsinki, Finland (IEEE 2012) [Selection rate: 19%]

• Invited Book Chapter

1. S.Al-Azzani and R.Bahsoon, *Architecture-Centric Testing for Security: An Agile Perspective*, *Chapter in Agile Software Architecture: Aligning Agile Processes and Software Architectures*, edited by M. A. Babar, A. Brown, K. Koskimies and I. Mistrik, (Elsevier 2013)

• To be submitted to the IEEE Transactions on Software Engineering

1. S.Al-Azzani and R.Bahsoon, *Architecture-Centric Testing for Security in Dynamic and Unpredictable Environment*, January 2014

1.5 Thesis Structure

The rest of the thesis is structured as follows:

Chapter 2 surveys existing research in security testing. The review explores the state-of-art and -practice in security testing and identifies gaps and need for architecture-centric testing for security. It explores how we can potentially leverage on the state-of-art and -practice to serve architecture level-testing for security.

Chapter 3 pursues the software architecture as the appropriate level of abstraction for evaluating, reasoning about, managing and facilitating the change and evolution of security testing. It advocates an architectural-based approach to testing as such is believed to offer the potential benefits of generality, scalability, as well as tracing and scoping the vulnerability analysis to low-level design artefacts. It recommends requirements, which are desirable to prevail when pursuing architectural level testing for security.

Chapter 4 describes a novel architecture-centric approach that exploits the concept of Implied Scenarios (IS) [185] to test the security of system architectures. It introduces the concept of Security ISs to deal with vulnerability analysis at the architecture level. It describes a three-phased approach for detecting and classifying threats, and using detected threats to guide the refinement of the architecture.

Chapter 5 introduce SecArch, an enhancement version of the IS approach with improved search-space. It uses detected race conditions to test for security with the presence of negative behaviour. It begins by drawing on comparison between complete vs incomplete models, and explores the benefit of merging both views to detect further emergent behaviour.

Chapter 6 introduces three applications and evaluation of IS approach and its extension, and reflects on the fitness of the approach to meet the requirements stated in Chapter 3. It evaluates the approach's applicability, scalability and effectiveness at detecting design-vulnerabilities.

Chapter 7 summarises the contribution and future work

2

Literature Review

Here, we review the landscape of security testing. The review explores the state-of-the-art and the state-of–practice in security testing, and identifies gaps and the need of architecture-centric testing for security. The review explores how we can potentially leverage the state-of-the-art and the state-of-practice to serve architecture-level security testing. One of the objectives we have is to share common understanding among security testing practitioners so that we can push the state-of-practice forward to build secure systems. Observations from the review confirm that current security testing trends are limited when addressing design-level vulnerabilities; in order to address these critical vulnerabilities, we need to shift focus onto the architecture/design levels to allow for detection of these vulnerabilities and prevention of their manifestation into the implementation. Efforts aimed at security testing at a high level of abstraction have been limited and lacked focus. Many of the identified works were scattered, with no unifying theme.

Parts of this Chapter are published in [6] and [155].

2.1 Security Testing: A Brief Background

Security testing is a process for determining whether a system contains vulnerabilities that can be exploited, and whether the software behaves and interacts securely with its users, other applications and its execution environment. The overall goal of the process is to reduce the number of vulnerabilities within the software system [118]. A vulnerability is a *weakness* in the system, which allows an attacker to violate the integrity of the system [153]. It may: (a) deny access to assets for authorized people or processes; (b) allow for privileged access to assets to unauthorized people or processes; or (c) allow unauthorized people or processes to hide assets [74]. The ways in which a vulnerability may be violated are collectively referred to as a threat scenario. A threat scenario is an undesirable behaviour of the system that, if executed, will result in an attack. An attack occurs when an attacker with a reason to strike takes advantage of a vulnerability in order to threaten an asset [153]. When an attack is successful, the security of a system is said

to be *compromised*. Compromising the system violates the security properties in effect, such as:

- violating confidentiality of data by preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information [98].

- violating integrity of the data and the system; data integrity is defined as 'the property that data meet on a priori expectation of quality' [98]. This covers data in storage, during processing and while in transit, whereas system integrity is 'the quality that a system has when it performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorised manipulation of the system' [98].

- violating availability; loss of availability may be caused by attacks (eg, abusing known system vulnerabilities) or instabilities of the system and its components [148].

Security Testing can be integrated during all stages of the development life cycle; it is performed using different methods at each stage to target different sets of vulnerabilities (eg, operating-level or design-level vulnerabilities). In the past, a great deal of attention was focused on implementation-level vulnerabilities, such as buffer overflows and invalid inputs, and general code level testing, while very little attention was given to design vulnerabilities [112]. This was the result of delaying the process of security testing to the later stages of development; many of the security vulnerabilities are architecture/design in nature [76], and thus code-level testing often fails to reveal them. When security testing is performed at later stages, significant changes to the design will often not be considered (due to timing constraints or financial costs [43]). Therefore, it is desirable to consider security early in the system life cycle while taking into account the partial knowledge of how and where the system will be deployed.

To allow for comparison between different testing methods, we focus on the following criteria:

1. Automation: the extent to which the methods support automation.

2. Vulnerability level: the vulnerability type the methods detect (ie, configuration, design or implementation level).

3. Support for observability: the extent to which they support observability. Observability in our context refers to how much of the internal behaviour we are able to observe from an external viewpoint.

4. Application Stage: at what stage is the method used (ie, design or implementation levels)?

5. The ability of the methods to support compositional security.

6. The level of expertise required for using the method.

7. Adaptivity to changes in development. It is understandable that anticipating all of the requirements of all users can be challenging. The ability to adapt to changing conditions is an important aspect of testing, which allows for ongoing verification that the system continuously meets its requirements and satisfies security issues.

These criteria are explained in the body of each method in details and how they are satisfied/achieved, as well in section 2.7 for direct comparison with the other methods. We summarise the current state of security testing to motivate the need for further research to address its limitations. Current security testing approaches can be summarised under six main headings, which shall be enumerated in the following section.

2.2 Architecture and Design for Security: Development & Evaluation

In this section, we review common approaches in developing and designing secure software systems.

2.2.1 Security in the end

In these approaches, security evaluation takes place after the design is built to determine potential problems in the design, either to refine or to derive test cases. This process is often done manually as in [125] and [48]. Mouratidis et al. [125] proposed a scenario-based testing approach that is geared towards testing a system at the time of the design. Their aim is to identify the goals and intentions of attackers in order to derive a set of possible scenarios for attacks on the system, and then test the system with these attacks to see how it responds. For the *threat detection* phase, their approach is best suited during detailed design. Another example is [148], where they use general security policies of an organisation to create a consistent security model that is used to derive generic security mechanisms. The main problem with this style is that system designs continue to evolve and, as the system evolves, security concerns need to be re-analysed in order to evaluate the impact of changes on the system [182]. This process of evaluating changes is believed to be challenging due to the ad-hoc way in which such approaches handle changes. To address this, an incremental approach is required to design security that is naturally adaptive to changes. That is, instead of having a specialised security phase for which security is evaluated, the goal is to integrate the security phase into the design choices so that it becomes an integral part of the design, which tends to incrementally change and evolve. Any changes considered are examined from a security point of view before they are enacted.

2.2.2 Model-checking Techniques

Modelling a system allows for simulation and analyses during the software development process, which can help developers better understand, manage and optimize the design [50]. Model checking creates a restricted model of the program that is expressed in some form of temporal logic to allow verification of correctness. Model checking is often used to detect inconsistency in the syntax and semantics of the system model. Examples of this testing for security include: 1) Jinhua et al. [86], who also integrate the concept of security properties and UML designs in order to model-check whether the design achieves the security properties or violates them; 2) At a higher level, Lodderstedt et al. proposed SecureUML [108], an extension of the UML language to specify security policies for role-based access control (RBAC). SecureUML is limited to describing RBAC policies, and thus addresses specific types of vulnerabilities. The same restriction was found in the works presented in [4] and [145], which also refer to RBAC policies; 3) A wider application of security UML models was proposed by Júrjens who presented UMLsec [89], a UML profile for modelling and evaluating security characteristics in order to guarantee basic principles in the whole system. The focus of this research is on building secure systems to guarantee a correct and secure behaviour, as opposed to detecting threats.

A subcategory to this is the *executable architecture model*, which represents the desired functionality of the system as a finite state model, while also modelling misuse cases to verify the presence of bad behaviour in the architecture. The importance of executable architecture is that it can reveal all possible communications [2]. Its advantage over other approaches is that it offers dynamic analysis to verify whether the set of allowable executions can divert from its intended behaviour, or whether it allows additional behaviours. Consider the following threat: A bank system that deals with two clients differently depending on the amount of money each client owns might reveal confidential information about an actor. In an executable or interactive model, this behaviour can be modelled as a branch, and is thus easier to identify than other various static models. An example of this approach, which was presented by Whittle et al. [192], used scenario-based spec-

ification language and extended interaction diagrams to model both desired and attack scenarios, where a synthesis algorithm transforms these scenarios into executable-state machines. Mitigation scenarios are then designed as aspect scenarios and woven into the core behaviour to prevent against the execution of the attack scenarios. Because they are added on top of the core design, they can be easily replaced or reused for different parts of the system. The core focus is on modelling threats and detecting whether or not they exist on the model, rather than using a model to search for potential behaviour beyond what is specified. In 2008, He et al. presented a similar scenario-based approach [71], with both an executable system and threat model. The process relies on software Inspections [151] (a manual process) for detection and validation of threats, and the focus of this work is on the use of data flow and trust boundaries (the degree of trust about a portion of the system) to detect violations of confidentiality, where secure data flows into untrusted areas within the system. Variations to state-based models were presented to enhance the test-case generation process, such as [197, 99, 11]. One important limitation in the application of this field for security is that it focuses on building the architecture model to be used as a means to verify whether a known threat is feasible in the architecture (where these threats are previously identified through other means such as threat modelling; see Section 2.2.5). This approach: (1) places an implicit trust on the behaviour of the architecture model; and (2) only checks what had previously been identified as a threat. However, as we will see in this thesis, security can benefit from executable architectures in a much more critical aspect of security testing, namely, detecting emergent behaviour in the architecture that is not initially anticipated by the desired specification. Using architecture models to extract counterexamples offers a very good level of observability, as these counterexamples can reflect complete internal/external behaviour. The counterexamples may correspond to the steps that an attacker can use to exploit the system vulnerability. A tester can then study the sequence of traces to analyse the security impact of any execution in the model.

2.2.3 Traceability and Conformance

Research including [137, 162], looks into ensuring that security requirements specified during the requirement phase are maintained in the architecture, such that the best-fitting architecture is selected with respect to the security requirements. In [137], Pauli et al. employs use cases/misuse cases to systematically identify potential malicious behaviours in the architecture. However, like many use case/misuse case-based approaches, the composition of functionality and its impact on security is rarely addressed at this level; instead, a component-based approach is considered in which each component is checked to ensure that it fulfils its role in achieving a security property (eg, privacy of data). We believe the reason for that is because the *traceability* of security requirements is often the focus of the transition phase. On the other hand, Shin et al. [162] limit this view slightly by associating components with the functionality of an application, and their connectors with security and communication between components, in which case components are not directly associated with ensuring their own security. The work of Oladimeji et al. [133] took a similar approach by adding tagged values to their architecture that model security details (such as SSH connection type); their focus is on determining whether or not a given software architecture model realizes a set of security policies; but like many of the surveyed works, they failed to address the requirement for architecture models to be amenable to changes in order to reflect new requirements and threats. The work of Antoun et al. [2] went further to support reasoning about security at both the architecture and code level. It requires intensive annotations of the code to build a hierarchical view of the system, as well as continuous refinement for the annotations. Along with the annotation overhead required to apply this approach, the focus is on ensuring conformance and traceability between the code and the architecture, where the code and the two architectures (one derived from the code, and one designed separately for comparison) are present in order for the evaluation to occur. Their security analysis is also property-based, where security properties such as 'well-formedness' (eg, two DataStores cannot be connected directly) are automatically verified.

2.2.4 Security patterns

The use of security patterns in designing secure architecture has received considerable research attention [75, 117, 103, 126, 156, 200, 195], after being originally coined by Christopher Alexander [8]. A security pattern is a well-understood solution to a recurring information security problem [146], such as a pattern that ensures secure authorisation or flow of data. It encourages effective reuse for building in robustness. Security patterns solve recurring problems in security [117]. Given a good pattern, it should be obvious to determine if the pattern is applicable to a particular situation or not, and how the actual instantiation of the pattern should be done. A pattern should clearly and specifically describe how it achieves the desirable solution, so that a designer can choose the right pattern and know what to do with it, and will understand the consequences of implementing the pattern. However, in practice, it is believed that these patterns are ignored in the industry [75], even though they are thought to be able to bridge the gap between the design phase and secure code. The reason for that belief is because a pattern should precisely describe its architectural design and how it can be achieved at an implementation level, possibly by using code snippets. If utilised correctly, they can be used to automate code generation.

2.2.5 Threat Model Driven Approach

Security threats are potential threats (ie, misuse and anomalies) that violate security policies or goals [76]. Threat modelling [192, 73, 172, 157] is a methodical review of a system design or architecture to discover and correct design-level security threats; the review determines an adversary's most likely course of action in order to develop appropriate mitigations [133]. This requires a clear understanding of the assets that are to be protected, the threat's objectives and any factors in the environment that could influence the threat's capability or decisions [80]. This process results in a threat model that describes the potential attacks on the system, which can be used to understand how attacks can

manifest themselves, and to evaluate critical decisions that will affect the security posture of the system [172].

Threat modelling is a sound approach for addressing software flaws at the design level [76]. Tondel et al. [180] showed that threat modelling is often considered an important part of the requirement phase as well as an iterative process, continuously revisited throughout the software lifecycle, while Microsoft had reported a 50% reduction in security vulnerabilities detected after incorporating threat analysis into the software development life cycle [64]. It is believed [166] that identifying threats assists in developing efficient and realistic security requirements for systems, where these requirements help to best identify countermeasures that defend against attacks. Others have taken another approach [148], where security requirements are initially specified and then used to determine mitigation methods or threats violating these requirements.

A common modelling specification for security threats is attack trees [157]. Attack trees precisely model the steps required to achieve a malicious goal in general, regardless of whether the application allows one or more of the steps [146, 109]. The root of the tree describes the target goal of the attack, whereas the sequence generated from a complete trace (from the root to a leaf) represents a unique attack that an attacker can follow. Attack trees have been implemented in a number of tools, such as SeaMonster [72], which was implemented by the SHIELDS project [159], and Suraksha [176]. They have also been researched intensively, and examples can be found in [179, 190, 109, 106, 99] where reusability of attack models has been considered. Other forms of modelling threats include activity diagrams, sequence diagrams [196], Petri-nets [195] and attack graphs [139]. Attack trees have also been extended to include more information useful for security [106], so that analysis and evaluation can be performed at the design stage for web applications.

Threat modelling has also been extended to fill the gap between threat-driven security requirements and high-level architecture design [196, 67]. Xu et al [196] extend misuses to allow variations of known attacks and exploits. UML sequence diagrams are used to describe the sequence of actions the attackers need to take to compromise the system;

these threats are identified through the use of the STRIDE threat model [73]. Then they derive the creation of architecture from the requirements such that they can assess whether or not the architecture is vulnerable to the identified threats before it is committed.

Threat modelling was further advanced to integrate attack trees with state charts in order to model lower-level dynamic behaviour [16]. This approach introduces systematic transformation rules and integration steps for mapping attack tree representations into lower level dynamic behaviour, and then integrates this behaviour into state chart-based functional models. Through the focus on both the functional and threat behaviour, software engineers can introduce, clearly define and understand security concerns as software is designed. The advantage of this work is that it takes threat modelling from purely threats into automated threat detection. Another interesting application [190] brought threat modelling into the implementation level to detect undesirable threat behaviour at runtime. Given the threat model in UML sequence diagram form that stimulates a threat scenario, one can track the runtime behaviour to see if it matches the traces of the threat sequence diagram model. If a threat is matched when exercising the code with random data, then the threat is believed to exist. This approach contains many weaknesses: The use of random data is not sufficient to prove that the system is free from the threats that are being tested, and the random test data may not successfully exercise the required threat sequence of the messages. It is labour-intensive testing because it requires the tester to keep modifying the code being tested to insert test statements that will be used to match traces. Another approach [157] uses the combination of attack trees and priority lists of assets [85] to show the scenario of the attacks in a visual manner that highlights which code, or part of the system, may come under attack.

Threat modelling has also been used as the basis for test case generation [109, 67]. In [109], the authors have proposed a threat model-based security testing approach that automatically generates security test sequences from threat trees and transforms them into executable tests. The security testing approach they consider consists of three general activities: building threat models with threat trees; generating security test sequences

from threat trees; and creating executable test cases by considering valid and invalid inputs. This approach uses the STRIDE threats produced by Microsoft to model and detect code-level threats. In [67], the authors use the architecture model for security testing, and in coupling security requirements with threat modelling for generating both security functional test cases and malicious test cases.

A capable modeller can think of various ways in which a threat can manifest into the system, thus the drawback in threat modelling is that it is entirely based on modeller's level of experience [190]. This process, which is the core body of threat modelling, is also not automated; thus, a change made to the design requires reanalysis of the system's security status and hence, incremental design changes –or even parallelism– are not generally supported [99]. Threat modelling addresses the design of a secure system, but not the methods for the testing phase, although the list of potential vulnerabilities may still be used as test cases. Additionally, if we utilize attack trees as an example for modelling threats, we discover that they lack semantic information about threats, and that they do not contain enough information about carrying out the attack [106]. On the other hand, observability of threats is a bonus in this approach. A tester can use threat scenarios to replicate behaviour precisely and understand the security consequence of its execution.

2.3 Model-based Testing

In section 2.2, we examined model checking at the architecture/design levels. In this section we look at the application of model-checking at the implementation level, where the difference is mainly based on the level of abstraction and the artefacts used for the analysis. Model-based testing is a variant of testing that relies on explicit behaviour models that encode the intended behaviour of a system and, possibly, the behaviour of its environment [141]. It consists of generating test cases based on the partial representation of the system under test, which also includes the specification of elements to be tested [140]. The advantage of model-based testing is that it allows for early test case generation

on abstract models [55]. It is also believed to support systematic security testing and reduces the level of expertise required for security testing, because its abstraction provides designers with a better understanding of the software under test. The complexity in building correct behaviour models is thought to be the reason behind its small use in practice, like in testing access control policies [55] such as [87, 142]. This belief is thought to be caused by the complexity of building correct behaviour models [185]. Once the behaviour model is built, verifying its correctness can be done via automated tools such as SPIN [156]. We briefly cover some of the existing work in this field as it relates to security testing.

In [124], the authors proposed a model-driven approach for specifying, deploying and testing security policies in Java applications. In their approach, security experts build a security model of the system under test using a security domain-specific modelling language; the security model captures access control policies defined in the requirements of the system. This model is verified and transformed into a policy decision point (PDP), where policy decisions are made. These decision points are inserted into the application code using: 1) aspect-oriented programming (AOP) and 2) mutation testing [39]. This approach is not only low-level code-based, but also depends heavily on security expertise, and addresses only testing access control policies. In [136], the authors present a model-based framework for security vulnerabilities testing. In their approach, they use three main models: one to describe the desired behaviour of the system, one relating to the behaviour of the various relevant components of the implementation and one for the attacker. These models are then combined together to generate test cases using a constraint solver. A similar work to this was presented by [138], except that they use stereotypes inserted into the original model without the need for generating an attack scenario model. Other approaches exist that uses MTB to test the application's security such as [104]. Model-based testing had also been used for testing cryptographic protocols [149, 37, 28].

2.4 Functional Testing

Traditional functional testing aims to verify that the system meets its functional requirements. A test case is successful if the functional behaviour exists and satisfies the requirements. Applications are tested to ensure that they behave as intended and do not allow harmful functional responses [20]. This is done on behalf of a legitimate user of the product who is attempting to use the application in the way it was intended to be used, and for its intended purpose. Functional testing is referred to as 'positive testing'. This concept is coined from the application's ability to alternately show or not show errors as appropriate. Positive testing corresponds directly to requirements and would dictate, for example, that 'user accounts are disabled after three unsuccessful login attempts'. On the other hand, 'negative testing' is focused on ensuring that the application does not do more than it is required to do; and so the requirement would be stated as 'unauthorized users should not be able to access data'. Such a requirement poses a lot of challenges for testers because, in many situations, the requirements cannot be mapped directly to specific software artefacts, and thus they are untestable in a traditional software development setting. For example, one cannot reliably enumerate the ways in which an attacker might obtain control of a software system. This shift in emphasis from positive to negative testing affects the way testing is performed. To determine success criteria for positive testing, one can enumerate the conditions under which the requirement being tested holds true, and verify that the requirement is really satisfied by the software. For negative testing, however, it is necessary to create every possible set of conditions, which are infinite in principle [55].

Security testing places greater emphasis on negative requirements, where the focus is on how a system is supposed to be, in contrast to what a system is supposed to do. Thus, it is important to test for situations that are not covered in the specifications to attest to the security posture of the system. This is because security, unlike functionality, is not an 'externally observable property' [163], and hence, one cannot easily predict its results or consequences. For software security, we aim to test situations that should not happen

(undesirable behaviour), and the functional testing may not be broad enough to address all possible ranges of malicious input. When security is addressed at the level of functional positive testing, focus is placed on verifying the behaviour of security mechanisms (such as authorisation checkpoints). This has two major pitfalls:

1. (Example 1: Firewall) Protecting an application using a software application such as a firewall proved insufficient, partly because: (a) the firewall may contain vulnerabilities such as buffer overflows, leaving the client better off without the firewall; and (b) if the protected application contains a vulnerability such as an SQL injection, then a standard firewall may not detect and prevent the attack.

2. (Example 2: Authentication) Testing for a requirement that states 'a login page must only accept correct passwords' may be tested with an incorrect password to increase confidence that the authorisation mechanism behaves as expected; however, this type of testing is insufficient on its own because it does not guarantee that an attacker cannot simply employ brute force to log in to the system. Verifying the correct behaviour of the system does not guarantee the absence of undocumented behaviours; it only verifies that the system behaves according to the specification.

The common types of functional testing for security are:

- Requirements-based testing [122], where test cases are associated directly with requirements to ensure that all requirements are satisfied. In security testing, the requirements are associated with the security properties of the system such as confidentiality and integrity of data. One of the most common methods for performing requirement-based testing involves *use* and *misuse* cases [164, 42, 40, 9]. Use cases define the requirements the system intends to achieve, whereas misuse cases define how the system can be misused based on the specified use cases; that is, they are the opposite of the use cases and represent the way in which that requirement can be misused. Security requirements are closely associated with misuse cases, and thus

these misuses help document the types of non-functional requirements [9]. For security testing, misuse cases have been used to model security threats and requirements, as well as exceptions and failure modes. Their advantage is that they help testers think past the positive aspects of the system. Our interest in misuse cases is not in their modelling capability but rather in their ability to support threat detection for test case generation. As their applications developed from modelling to detection, some research [134, 40] compared misuse cases to attack trees [157] (another modelling technique) to evaluate the ability of both approaches to guide the threat detection process. Other researchers [123] looked at using these requirement-level modelling techniques to derive injection test scenarios in order to test the security properties of the protocol under evaluation. Tondel et al. [179] looked into linking misuse cases and attack trees to obtain a high-level view of the threats towards a system through misuse case diagrams, and established a more detailed view on each threat through the employment of attack trees. The work produced by Karpati et al. [93] and [137] took the application of misuse cases further by integrating security threats into the architecture, supporting the transition from requirement specifications to high-level design, and vice versa. The drawback in [93] is that it requires creating a detailed map of the vulnerability. Misuse cases are also useful for trade-off analysis [10], the goal of which is to enable stakeholders to make an informal and correctly based judgement in a potentially complex situation. Hybrid techniques have also been introduced to complement the limitations in both misuse and attack trees for the elicitation of security requirements [63].

- Specification-based testing [169, 193, 1, 31], in which test cases are derived from specifications such as interfaces. An example of this is the method produced by Blackburn et al. [26] for testing database systems, which automates security functional testing, given that all variables, data-types and security functional requirements of the system under test are modelled in an algorithmic language. Another widely-applied specification-based testing is that used by the access-control policy

community to generate test cases for security policies (SP), such as [181]. Traon et al. determine their security policy from requirements. Once the policy is determined, they propose different strategies to generate test cases from a security policy model, with the purpose of obtaining evidence that the SP implementation is correct to the requirement. Another similar work presented by Scott et al. [158], where they abstracted security policies from large web applications to verify the correctness of firewall policies. Similarly, Bracher et al. [29] attempt to test the security of protocols by generating test cases from an 'intermediary model', which is a less abstract model of the protocol as compared to [158].

- Another widely accepted approach for functional testing is state-based testing. This field is relatively large as it incorporates property-based testing. Examples of this are those proposed using UML state-charts such as [99, 11, 129, 97, 156], where the purpose is to model the functionality using state-charts, after which constraints are created to verify whether the security properties are violated. An example of a security property could be that 'unauthorised access leads to the activation of countermeasure' (ie, reaches prohibited state in the state-chart). This method is slightly more advanced in that it exposes potential design flaws between security controls and functional behaviour (eg, operations with no permitted access).

In summary, the main disadvantage with functional testing for security is in the focus and goal of the testing process. Verifying correct behaviour of security mechanisms is not the main focus of security testing, as discussed in Section 2.1. There is also a general limitation in that there is an unfortunate separation between the requirements reflected in the test cases [91], which may contradict in some situations, leading to gaps in the detection of design errors and inconsistencies. Verification of the compositional security status is often overlooked in these types of approaches, because it requires looking beyond the specifications. It is not only enough to ensure that function A and B work, but it must be understood how function A and B work when composed together with respect to potential race conditions or any additional behaviours that may arise.

As we have broadly seen, the variations of functional testing allow it to be integrated into different stages of the development cycle; for example, state-based testing can be used at both the design and implementation levels, and thus supports the detection of vulnerabilities at both levels. Moreover, automation seems to be the way functional testing is performed, often at a unit and integration testing level during the implementation phase. Achieving full automation when checking the correctness of behaviour is possible since conditions can be checked against, but this is not always possible for security; for example, performing brute-force testing on an authentication mechanism using test suite 1 may not give the same result as test suite 2. The challenge for functional security testers becomes focused on 'what conditions can we test to verify correctness?', which is by far easier than 'what possible behaviours can break this mechanism?' because the answer to the latter can be infinite. This is where observability can be a problem because, for security in general, observation of the entire internal state can be a serious challenge, even for small systems. At the level of adaptivity, functional testing tends to be relatively simple, because the focus is on changing requirements that are directly reflected by test cases. In other words, adding new functionality, will only require adding new test cases, and removing functionality will require removing related test cases.

2.5 Penetration Testing

Penetration testing is the process of exercising an existing application using predefined test cases that simulate an adversary's attempt to achieve a malicious goal [190]. It is the traditional security approach that aims to verify that the system is protected against the penetration test cases. Institutions such as the Institute for Security and Open Methodologies (ISECOM) published an Open Source Security Testing Methodology Manual [74] that describes in detail how to perform thorough penetration testing. It targets vulnerabilities within the environment/configuration on which the system is running, and often takes the form of black-box testing of the system, using a predefined set of test cases that

represent known exploits. It is performed using either existing tools [173, 17] for full automation, or as an ad-hoc and exploratory testing [83, 44] in slightly informed greyed-box settings, whereas test cases are derived based on the tester's skill and experience with similar programs. The kinds of activities involved in penetration testing range from host address spoofing and network sniffing [143], OS finger printing [175] and brute force and vulnerability scanning/analysis [139, 165, 100]. Penetration testing also varies between external testing, where the system is penetrated from outside the organisation both with and without disclosure of the system, and informal testing strategies where penetration is performed from within the organisation to determine what an insider might to do the system [198].

Penetration testing comes into play in the final stage of development when the application is fully implemented, and although redesigning a system is possible, the cost of doing so sometimes outweighs the advantages gained in the eyes of the developers; patching a system is often cheaper and is likely to be considered before redesign. Patching is closely associated with penetration, in which the system is first penetrated for problems, and then patched for correction. This step attempts to hide the symptoms of the problem as opposed to fixing it, which may bring many issues into the system, such as writing a vulnerable patch or discovering new symptoms of the problem. This also means that, if we detect a serious design flaw that requires redesign, testing of the changes will only occur after the system is implemented, which greatly hinders the adaptivity to changes associated with this type of testing. Although this approach is attractive, its main problem is that it is only effective when performed by trained or experienced testers to design special tests not captured in more formal techniques [118]; it is also not design-specific testing, since testers often run the same set of test cases on different systems, relying on the fact that developers often make similar mistakes and repeat them. Passing a given set of tests does not imply that the system is secure, nor does it imply that the system will pass any given security test. In addition, the penetration test suite needs to be continually managed and updated to render effective testing for security. For example, as network

technologies develop, new test cases will need to be added to reflect new threats, and old test cases may become obsolete. This management of test suites requires augmentation, omission and the checking of the quality of the test cases in achieving the required coverage.

Security tools used in penetration testing such as ISS Scanners [173] and Cybercop [17], are generally limited in scope [45]. They mainly address network security attacks, and are not flexible enough to allow testers to write custom attacks. Another problem with the existing tools is that they can only be used after the system is built. In addition, most tools address IP networks, and so a company wishing to test a different type of networks is required to purchase different tools as required. Although these 'badness-omiters' [111] are useful in displaying the negative state of the system, especially when the system configurations are well understood, they are not useful in non-standard applications, and hence, should not be the only method for testing an application. Other form of security tools are static analysis tools that address code vulnerabilities such as buffer-overflow. They are very limited in scope since dynamic testing is also important, and they have high false-positive error rates. A survey was conducted to test various security tools [45] concluded that tool are not adequate without manual testing and human judgement.

In addition, penetration testing, whether done by hiring a red-team or using network vulnerability scanning tools, addresses known attacks without a hope of finding new ones, and determined attackers often look for novel ways to trigger unspecified behaviours in the system. The response of the system to such attacks is observed and any inappropriate behaviour is noted. Observability is a serious problem here, because testers observe a response from a black/grey box, which means that, unless the system displays an error or gives some form of response, the internal state is not fully investigated, and thus malicious states can be easily overlooked. This process requires the knowledge of both the desired behaviour and certain implementation details that are the source of vulnerabilities [136].

To summarise the problem with this common approach [114]:

- Only known problems can be fixed. Attackers do not report vulnerabilities to developers.
- Because it occurs late in the development cycle, significant changes to the design will generally not be considered; thus, patches only fix symptoms of a problem and fail to fully address the causes, especially if the problem is a design defect [68].
- By the time a vulnerability is discovered and a patch is made available, attackers would have enough time to compromise the system, leaving system administrators one step behind the attackers.
- The current ad-hoc manner in which vulnerabilities are patched ensures that many Internet sites will remain vulnerable.
- The patch may introduce new vulnerabilities because it is often programmed under intense pressure and is not adequately tested.
- Patched code often does not make it into a subsequent version of the software, resulting in the reappearance of the same vulnerability in future releases.

2.6 Fault Detection in Security Testing

In security, a fault corresponds to an inconsistency between what is desired to be achieved, and what is actually achieved during runtime [181]. This inconsistency may cause a reduction in, or loss of, the capability of a functional unit to perform a required behaviour [19]. As opposed to traditional testing, fault detection focuses on the unexpected [38]. In the field of fault detection and tolerance, a common approach to fault detection is performing 'information redundancy' checks, where two operations run in parallel, and when the results are compared and found to be different, then a fault is reported. Not only is this type of testing exhaustive to resources, as errors might not arise until very late in the

process, but there is also an assumption that the original process is correct by definition, which is not useful for security testing, as the original process might be vulnerable, and so trusting its outcome is questionable. In addition, process-based testing is limited, and further research on compositional problems across processes is required.

Fault detection in security often takes an attack-driven approach. One method is to modify the system model to introduce errors that can be exploited by attacks [55]. Such changes to the system model are referred to as *mutations*, and are used to simulate attack scenarios in order to provide observable responses to the attacks. During the simulation, a tester executes a test case that corresponds to an attack against the system model, and makes a note of the observed response; this observed response is later used to confirm whether the test case executed against the actual system model triggered a similar response. If the system behaves as the modified vulnerable system model suggests, then a vulnerability is found. Test cases are generated that focus on the introduced errors [39, 89]. This approach is significantly important in that it supports observability of the problem, which is thought to be difficult to achieve in other tests, such as functional testing. The challenge here is caused by the fact that we may not always receive a response (eg, error or wrong input) to the test case. However, the difficulty in such testing is that prior full knowledge of attacks and their likely trigger points is required, and the set of possible attacks on the system must be understood. This type of testing is heavily utilised in testing the implementation of cryptographic protocols [25, 18, 13, 127, 54], networking [77] and in access control testing [181], in which the mutation of policies can be easily identified; for example, a test case used to request access to secure data, where the success of the data access represented a failure of the access control [54]. At the code level, MUSIC [160] and MUTEC [161] tools have been developed for mutation testing to detect XSS- and SQL-injection attacks against web applications.

2.6.1 Fuzz testing

Fuzz testing is another technique for performing fault detection [119]. As opposed to muting the system model, the very basic idea behind a 'fuzzer' is to test a protocol implementation for possible security flaws arising from the improper handling of malicious inputs [55]. Fuzzing complements traditional testing to enable the discovery of untested combinations of code and data by combining the power of randomness, protocol knowledge and attack heuristics [38]. Fuzz testing can take the form of black-box testing, in which a random mutation is performed on input data that is used to exercise the system. Empirical studies [88] have proven its effectiveness in revealing vulnerabilities of software systems. According to Takanen [174] and Yang et al. [197], it is believed to have created a quick, automatic and cost-effective method for finding critical security bugs in large applications. It also covers a significant portion of security *negative test cases* without forcing the tester to deal with each specific test case for a given boundary condition [131]. If, for example, we have an input that should be between 1 and 10, a boundary fuzz testing would generate the test cases that include 0, 1, 11 and 12 as input test data. However, the current effectiveness of fuzz testing is limited when testing applications with highly structured input (eg, checksum); randomly testing applications usually provides low code coverage. These limitations mean that potentially critical security bugs may be overlooked.

On the other hand, the performance of fuzz testing under white-box conditions requires detailed implementation knowledge about the system under test. These fuzzers are referred to as *intelligent* fuzzers because they are able to dig deeply into the code to perform targeted testing. This addresses limitations in traditional fuzzers, which suffer from randomness, by providing a structural approach to fuzzing (eg, (1) Yang et al. [197] presented an approach for combining the idea of fuzzing with the concept of model-based testing, to allow for systematic and automated testing of software applications, (2) Sodiya et al. [166] presented an approach for the fuzzification of input variables that is based on the six major categories of threats (STRIDE) [73], rule evaluation and aggregation of the rule output). The negative aspect to this is that fuzz-testing tools must be built (or

at least configured) for each file format and/or network protocol that is to be mutated, in order to create to create a semi-valid format [38]. If we take encryption as an example, the fuzzer must be able to decrypt data before it mutates and then re-encrypts. To address some of these limitations, Dai et al. [38] presented a configuration-based fuzzing approach. Instead of generating random inputs that may be semantically invalid, configuration fuzzing mutates the application configuration in a way that helps valid inputs exercise the deeper components of the program under test, and checks for violations of security requirements, such as 'a user should never gain access to files that do not belong to him' or 'critical data should never be transmitted over the Internet'.

The literature we surveyed on security fault detection seems to be drawn on the implementation level, with a focus on implementation level vulnerabilities. As a result, the approach tends not to be very adaptive to changes as changes to the model (eg, changing a cryptographic protocol), which require a possible redesign of the fuzzer. However, there have been attempts made to employ fault detection at the modelling level [109, 90], though additional work is needed to bring the field to maturity.

2.7 Discussion and Summary

In this chapter, we have seen and reviewed a number of existing methods for security testing. We have discussed these methods with respect to our comparison criteria, which are highlighted in Section 2.1. Table 2.1 gives a relatively generalised comparison of the majority of papers we have reviewed in the compilation of this literature review. These methods address security at different: 1) stages of development; 2) sets of vulnerabilities; 3) levels of required expertise; 4) levels of automation; 5) observability support levels, and finally 6) they demonstrate how adaptive this method is to changes that occur during software development. We have taken care to state when the generalised comparison does not apply to over 70-80% of the reviewed paper. In this situation we have used the term 'often' in the table to reflect that difference.

As discussed in previous sections, we argue that architecture-level testing for security shall satisfy the following criteria (see Section 2.1 for criteria). Below we discuss the limitations of existing work in relation to these criteria.

1. (Adaptivity = high): The reality of the development cycle is that requirements are changeable. Whether new requirements are added or deleted, the approach must respond to changes incrementally and iteratively to reduce processing power with respect to re-analysis, and to avoid ad-hoc approaches in responding to changes (eg, manually identifying areas that need to be reviewed after changes occur). There are two important aspects as to why adaptivity is required: (1) security testing is a process that is likely to introduce changes into the design in order to make it secure; (2) as requirements change, security vulnerabilities may be introduced into the system. As a result, an approach is needed that would allow us to adapt to changes as they occur, as well as to examine these changes to ensure that we do not introduce new vulnerabilities into a secured design.

2. (Automation = medium): Full automation in the use of security evaluation tools (as discussed in Section 2.7) seems to have failed, except in a few cases [183, 45], either due to the number of false-negatives/positives, or due to their limited scope. On the other hand, manual tasks put a lot of constraints on security testers especially for moderate sized applications. It is thus necessary to devise a semi-automated approach that can balance between the necessary automation (to reduce time and cost) and human judgement, because hacking is a creative process, and creativity cannot be easily automated. In order to get the most out of security testing, minimize costs, and work more efficiently, we need to bridge the gaps that exist between automated testing and manual testing. This is arguably the most important step in making big changes to the security testing approaches.

3. (Level of Expertise = medium): According to Microsoft [64], 64% of developers are not confident about their security skills; thus, providing approaches that rely entirely

on the expertise of security testers may fail in practice if testers are unqualified for their use. Since performing security testing is believed to be relatively difficult [177], it is therefore important to invent new approaches that provide a level of support for security testers (eg, providing some form of clues to where threats may be located, and suggesting which areas are likely to be more problematic).

4. (Observability = high): In this chapter, we have discussed that black-box settings allow the observation only of responsive functions (eg, error messages, wrong outputs, etc), but security is not an externally observable property; thus, even if the correct error message is displayed, other behaviours inside the black box might not be observed. A supportive feature in testing approaches involves exposing the necessary internal behaviour that could aid the tester in determining security breaches. In large applications, exposing the large internal state is relatively difficult, if not impossible. Abstraction is vital to achieve a reasonable set of observable behaviours that is related to the set of behaviour under test. This allows us to study what further behaviour can occur beyond that which we can perceive externally.

5. (Stage of development = early): The stage of development for which an approach is used is closely related to the adaptivity of the approach. When an approach is introduced at late stages (maintenance and testing), responding to recommended design changes could be costly. This is one of the reasons why a 'penetrate and patch' testing approach is widely adapted in the industry. Late stages are also prone to tight schedules leaving a small amount of time to be dedicated to thorough testing; additionally, the amount of details present at late stages can be over-whelming, creating a desire for abstraction, where attention is focused on the most critical parts of development. Because we believe that abstraction plays a major role in efficient testing (because it captures only those details about an object that are relevant to the current perspective), the architecture-centric approach provides the necessary level of abstraction to assure both a decent level of adaptivity as well as

the support required for design-level vulnerability detection.

6. (Vulnerability level = design): In section 2.1 we have looked at and discussed the need to invent new approaches that targets design vulnerabilities, and that the current approaches seem to draw more focus on implementation level vulnerabilities. We have highlighted that design vulnerabilities are more critical because they often cannot be solved at later stages of development without serious cost in system re-design, and that they tend to have much greater impact in terms of exploits and security consequences [12].

Throughout the discussion for each category, we have explored the strengths and limitations of security testing; none of these methods seem to precisely fit the requirements stated above. The closest approach is high-level testing. Despite its ability to offer a good design-specific approach, changes to the design can be adaptable, and so these methods still do not meet the desired requirements. Either they require a high expertise level to support design-vulnerability prediction (and so feature low automation), or where the expertise level is low, with thigh automation levels and a focus on the detection of code-level vulnerabilities. The reason for this is because design-level vulnerabilities are thought to be hard for automated tools to detect. This emphasizes the need for an architecture approach that offers a medium level of automation to support a speedy process, while maintaining a medium level of expertise to balance the workload for an efficient testing process. The literature we have surveyed on high-level processes indicated that the focus is often on code level vulnerabilities such as XSS and SQL-injection attacks which in practice require detailed designs (eg, flow of data). Additionally, this field is immature with respect to compositional security and the ways in which interconnected functionality can cause unexpected behaviours; we find that testing for security in this field model requires individual functionality and related aspects to that functionality. This limits the observability behaviour in that we can only observe internal cohesive behaviour, as opposed to observing the functionality, with respect to the ongoing environment of composed behaviour. Furthermore, iterative approaches are generally lacking for security,

because security is often handled at a higher level as one phase of the process rather than as a continuous and evolving property. In Chapter 3, we will discuss in greater detail the importance of this with respect to security.

Table 2.1: Comparison of different approaches used in security testing. The comparison uses (1) adaptivity to design changes, (2) its support to automation, (3) the expertise-level required to carry-out the approach, (4) the extent to which the approach is observable, (5) at what stage the approach can be used, (6) vulnerability types detectable, as comparison criteria between the approaches.

Testing Methodology	(1) Adaptation To changes in design	(2) Automation	(3) Level of Expertise	(4) Observability	(5) Stage of Application	(6) Vulnerability-level focus
FT of Security	Low due to its late application	High	Low: Simplified because the desired functionality is known	Low: Often performed in black/grey box setting	Often: Maintenance	Code-level
* Req-based	High	low	low	N/A	Early-stage	high-level
* Model-based	High	High	Medium	high	Early to Maintenance	Design and code level
Penetration Testing	Low due to its late application			Low: Often performed in black/grey box setting	Testing phase	Configuration level
* Tool-based		High	Low			
* Expert red team		Low	High			
Threat Modelling	Medium: May suffer from ad-hoc changing process	Low	High	High: Has the advantage of focusing the study on the consequence of an attack over the whole system	Often: Design Stage but is also found at later stages	Design and code level
Fault testing					Often: Maintenance	Often: Code-Level
* Attack-driven	Medium: May suffer from ad-hoc changing process	Low	High: Requires knowledge on where to insert errors	High: Internal behaviour is observed		Often: Code-Level (Can find design level errors if performed at early stage)
* Fuzzers	Low: Requires re-design of fuzzers to respond to changes	High	High: Requires design of fuzzers	Low: Often performed in black/grey box setting		
High-level	High				Often: Design stages	Often: code level, but is capable of finding vulnerability
* Exe-Architecture		High	Low: Tools are often used to generate executable architecture	High: Exposes internal state of behaviour		
* Separation of Concern		Low to Medium	High	N/A		

3

Requirements for Security Testing

In Chapter 2, we examined approaches for post-implementation testing for security. We discussed the negative implications of their late usage for security testing. Furthermore, many of the approaches appear to be ad-hoc and lack discipline when dealing with large amount of code and configurations. We pursue the software architecture as the appropriate level of abstraction for managing and facilitating the testing process for security and for refining and evolving secure software systems. We advocate an architecture-based approach to testing as this is believed to offer the potential benefits of generality (ie, the underlying concepts and principles are applicable to a wide range of application domains); an appropriate level of abstraction to tackle the complexity of the problem (ie, software architecture can provide the appropriate level of abstraction to allow for the consideration of secure composition and interaction); the potential for scalability (ie, facilitating the testing for security in large-scale complex applications); the opportunity to facilitate automated analyses; and the potential for tracing and scoping the vulnerability analysis to include low-level design artefacts. Guiding the testing for security through architectural and design artefacts stems from the belief that most vulnerabilities tend to be architecture- and design-based in nature. In the coming sections, we will look at the main requirements for bringing security testers ahead of attackers when developing and continuously refining architectures of software systems.

3.1 Proactive Approach

Proactivity is defined as 'having an orientation to the future, anticipating problems and taking affirmative steps to deal positively with them rather than reacting after a situation has already occurred' [61]. In the state of practice, there is a belief that we are often (at least) one step behind of the attackers [3, 201, 184], and so we focus on fixing what had already been exploited (eg, reviewing Cert's advisories and taking the appropriate actions); even in research we find that focus is given to defence mechanisms that target newly created attacks [58]. This reactive approach is simply learning from the past,

which is far easier than predicting the future. But with the increasing number of computer security incidents reported daily, and the speed at which they are exploited, it is no longer sufficient to be reactive; a reaction to an attack might be too late where serious damage has already taken place (eg, theft of bank data), or where addressing the problem might be too costly if a redesign of the system is required. In addition, while attackers have all the time to choose when and where to launch an attack (which minimises their cost), security testers must respond fast and within narrow time constraints. This cycle of chain provides an advantage to the attackers; thus, testers must break this cycle by anticipating the moves of the attackers, and by attempting to remain one step ahead of them by identifying emerging vulnerabilities. Because the landscape of threats is constantly evolving, security testers must learn to think like attackers, following clues to insecure behaviour and exploring potential vulnerabilities, because attackers intentionally probe unspecified behaviours in the system; therefore, testers must take a proactive approach to detecting weaknesses and predicting their possible occurrence. Proactivity calls for taking the lead in preventing the occurrence of vulnerabilities, and when they occur, fixing potential vulnerable states in the system by means of redesign, where we build security into the application to minimise the need for reactivity. This provides the opportunity to assess risks of potential vulnerabilities and potential design mitigation tactics prior to the implementation and deployment of software.

If we consider the process of code reviewing (where code is assessed for implementation vulnerabilities before deployment), we discover that it offers a level of proactivity because testers assume the attacker's position in searching for problematic lines of code. However, they are primarily focused on known threats (what we have learned from the past), excluding yet-unknown vulnerabilities from assessment, and therefore the associated risks remain unmitigated and unpatched. This becomes more problematic when the problem is design-specific, and thus may not benefit from other application weaknesses. It then becomes the tester's responsibility to violate these assumptions in an attempt to uncover vulnerabilities. Security testers must consider actions that are outside the range

of normal activity - tests which might not even be regarded as legitimate under other circumstances. We thus argue that new approaches must explore potential weaknesses that are design-specific and offer a decent level of predictability with respect to possible behaviours of the system. In Section 3.3 we will look at the importance of design- specific approaches in more detail. We conclude that proactivity plays a major role in designing and implementing secure systems, and is therefore a requirement for an efficient security testing process. We need to devise an approach that helps testers in the determination of potential vulnerabilities, and assesses their impact if successfully executed; in other words, an approach that identifies the possible negative operational effects associated with a successful attack.

With reference to the definition of proactivity, we consider the architecture as the appropriate level of abstraction at which proactivity can be fulfilled, where it: 1) enables design defects to be discovered early so they can be mitigated in a cost-effective and timely manner before the attacker exploits them; 2) enables building security into the system during its core design decisions; and 3) provides the knowledge base that paves the way for predicting the security impact of proposed design changes and future plans to evolve the system [23].

3.2 Architecture-based

In Chapter 2, we examined many post-implementation-level testing approaches, and we have explored their weaknesses in terms of late application to the testing process, and owing to the fact that many approaches appear to be ad-hoc-based because of the large amount of code and configuration that had to be dealt with in a short, constrained time. To address many of the issues associated with late application of security testing, it is believed that software architecture can play a vital role in the development of secure systems [163], because it supports the view of the system as a whole, and visualises all possible entrances/exits to the system resources and components. This addresses the scalability

and complexity of systems, enabling us to omit parts that are not necessarily important for security testing, thus focusing the attention on interactions that are particularly vulnerable. Object oriented systems, for example, are thought to be complex and tedious [101], especially for large applications. A contributing factor may be that testers become overwhelmed by the information such that understanding the intended behaviour of the system under test may not be guaranteed. Developers build a certain mental view of the software, and that mental view is limited because the software is too complex for the human mind to completely picture. Abstracting the information offers better opportunities for efficient testing, especially in terms of predicting undesirable behaviours, which is what is required to remain one step ahead of attackers.

Additionally, the architecture plays an important role in realising the use of third-party components. Since it may not be viable to modify the source of such components (either because the code was shipped in binary form or because the license agreement is prohibitive), it is not obvious how security vulnerabilities could be detected at the coding level. But, if we are able to observe its behaviour externally through its communication with other components, we are able to detect undesirable behaviour and it is here where executable architecture models can make a significant difference in observing and predicting the behaviour of components. The purpose of modelling the architecture is to enable the prediction of the system behaviour. To accomplish this, we need ways to study the effects of interconnected components on the preservation of the global security behaviour. It is useful to replay component behaviours in order to visualise the dynamics and consequences of desirable behaviours, so that if the model deviates from the intended behaviour, the design of the system can be modified early at a minimal cost. Dynamic architecture analysis is concerned with demonstrating software-predicted runtime behaviour. It is a necessary part of testing as it supports analysis with respect to operation environment variables. Not only will this permit predictability of malicious behaviour, and help testers understand the desirable behaviour of the system, but it also supports test case planning and generation at early stages of the software development

life cycle, supporting the processes of coding and testing in parallel to address the problem of testing being performed under immense time pressure. Test cases can be derived from requirements regardless of the programming language or technology utilised, which allows for the testing of existing systems. Testing a live application to achieve this result at the code-level is very expensive and time-consuming. Thus, we find that modelling the system at the architectural level is a cost-effective approach, which permits testing of the security behaviour of the system, reduces the cost of testing and turns it into a more mature and considered process.

In the rest of this chapter, we will reflect on why the architecture helps in achieving the rest of the requirements.

3.3 Design-Specific

In software engineering, security by design means that the software has been designed from the ground up to be secure and to minimise the impact of system breaches when a security vulnerability is discovered. In order to achieve security by design, it is necessary to take into account the individual design of the software under test. Current methods use pre-determined vulnerability databases to discover known vulnerabilities in a system. However, each application has its own design, and each design has its own weaknesses; therefore, each application, and its design, is unique and must be tested individually. This is motivated by the fact that programmers often make the same mistakes. But the assumption that we can run a set of known and common test cases to provide sufficient security testing is relatively unrealistic, because security testing is motivated by addressing undocumented assumptions and areas of particular complexity to determine how software can be compromised [80]. Existing techniques fail to address implicit assumptions, mainly because testers rely on the system specification and code [118]. An example of this is design mismatches in which a component makes assumptions beyond what the environment can provide, and attackers intentionally probe unspecified behaviours in the system

[177]. They attempt to make the application behave unexpectedly, and then determine the attacks associated with that behaviour.

Design vulnerabilities are a subtype of vulnerabilities that are associated with design defects and implicit assumptions, which cause security violations; an attacker can compromise the system if he is able to interact with the system in ways that were not anticipated by its designer. Avoiding design-level vulnerabilities is one of the most important challenges facing today's software developers [146, 194]. The fundamental importance lies in the fact that even the most secure implementation cannot guard against defective design. An example of a design vulnerability from a recent case study [53] allowed ciphered text to be accessible to the user to provide a sense of data security. This was identified to be a major design vulnerability, as it permits the user to cryptanalyse the ciphered text to identify the cryptographic key used during encryption. These design vulnerabilities cannot be easily identified using implementation- level testing methods; detection requires a design-specific testing methodology. In addition, they cannot be addressed at later stages in the development cycle unless the system is redesigned [190]. These design vulnerabilities tend to have much greater impact in terms of exploits and security consequences [12], as revealed by the statistical data cited in Chapter 1. In fact, one of the leading problems in security testing is that security testing is often not integrated in the design phase of the system development; as a result, fixing design vulnerabilities can be costly, which leads to the common choice of 'find and patch' in security testing. Patching is a process of hiding symptoms of the problem, as opposed to fixing its root causes and correcting the design. In many situations, patching does not offer a solution to design vulnerabilities; additionally, it is also difficult to enumerate all symptoms in practice in order to prevent the vulnerability from being exploited. This emphasises the need to bring security testing into the early design stages of development so that design vulnerability can be addressed before the system is built.

To address design vulnerabilities, it is necessary to work at the architecture and design levels to minimise the impact of design defects. A design defect introduced by incorrect, inconsistent or incomplete architecture design decisions will propagate throughout the refinement of the design, and is therefore likely to have a deteriorating impact on the complete development process. From a security perspective, experts consider architecture and design to be the single most critical phase of the secure design life cycle [12, 163], because good decisions made during this phase yield an approach and structure that are resilient and resistant to attacks. Unfortunately, the importance of architecture-level testing for security has only been recognised recently [113] for defending against design vulnerabilities, which are believed to be the hardest type to find and correct, and are also the most critical to address.

3.4 Iterative and Incremental

Despite the existing security development life cycles (such as Microsoft's SDL [79]), which aim at integrating security into development, the focus is mostly on enforcing certain check points (such as risk analysis or threat modelling) after certain phases of the development have been accomplished, rather than on devising a test-driven approach to guide the design choices incrementally as they are made. For example, a pre-defined design to perform threat modelling would require revisits to a set of design choices that may not be easily modifiable at the time without refactoring the design (eg, if we made subsequent design choices based on them), and even when these changes are performed, a re-evaluation would repeat this process as new changes occur. Continuous refinement, as the design is built, is more efficient than performing one stage of refinement process at the end of each design phase, because this ensures that the design is built with security in mind, rather than being evaluated for its security after the choices have been made and adopted.

Given that the nature of software development is incremental and iterative, it is important to devise a testing approach that is adaptive in nature to support software evolution. When changes occur (such as changes in requirements), new vulnerabilities may be introduced. For example, adding a new functionality in an update may require an elaboration of user access rights, which in turn may lead to new security measures and use cases (eg, keeping a log of who used the functionality). This process of making changes and continually revising the design to ensure it remains secure, is relatively challenging to achieve. It would be desirable to assess the potential consequence of changes (and whether the changes are advisable) before they are committed into the design. This would allow the testers and designers to make informed decisions about the security of the evolving system, and as a result, maintain/enforce a secure design.

The ability to watch the evolution of requirements means addressing ambiguity in requirements. This is because requirements often come from different stakeholders, who may have different understandings of the desired behaviour, and thus offer the opportunity for stakeholders to periodically see the progress and make adjustments to the requirements as needed. This early feedback provides knowledge on the resilience of the software to predict security threats so that changes are made before the system is built, and also responds to changes in user and behaviour requirements, which could impact the security of software. The nature of architecture offers a high potential for incremental and iterative development, given that it offers an abstract view of the system and is used at early stages of development. However, the challenge in securing an architecture is that the architecture must not only address security issues but also be flexible and resilient under constantly changing security conditions. For example, the longer duration of data preservation means some of the security codes and policies may change, and security requirements will have to evolve over time as user requirements change. Thus, it is necessary to incrementally incorporate security testing into the refinement processes of the architecture and design, while taking into consideration the incomplete knowledge of the architecture in the early stages of development.

In addition, the chance that a design vulnerability will be exploitable is very small, because testers will be constantly revising the results for any potential problems. Thus, even if the vulnerability passes undetected in the first iteration, the chance of its detection after several refinement rounds is high. This concept of iteration verifies that the system architecture adheres to requirements and stakeholder needs and, most importantly, to software security. This cyclic process is dependent on time and budget constraints. In certain circumstances the cost of continued testing does not justify the cost to the project, such as when: (1) the process may not reveal any new critical information (ie, the risk had been minimised); or (2) previously identified information is continuously repeated (ie, there is no apparent variation in the behaviour of the program), or there may not be any further emergent behaviour detected.

3.5 Secure Compositions

A general ability to build composite high-assurance systems presupposes a general theory of system composition [115]. Such a theory provides insight into why certain security properties are preserved/not preserved by certain forms of compositions. In other words, if functionality F1 satisfies confidentiality, and functionality F2 satisfies confidentiality, will the composition of F1 and F2 preserve the confidentiality property? This uncertainty might be triggered by changes in the functional requirements for which new compositions are needed, or by changes in the environment in which the application will be running. In the 1980s, McCullough defined a secure system based on the compositional factor of development as 'A system is secure if it is deducibility secure and if, when it is hooked up with a second secure system, the result is a hook-up secure composite system' [110]. The challenge in achieving compositional security is that security is a global property, yet many big systems are built using smaller components gathered from different sources. Given a set of specifications, we need to verify that all relevant components collaborate to ensure that global security of the system is achieved. Unfortunately, most vulnerabilities arise

from unexpected interactions between different system components [144]. This means that if we study the behaviour of every individual functionality to ensure that it operates as intended and is secure, this will often still not guarantee that the composition will lead to a secure overall behaviour. Thus, given valid functionality traces T1 and T2, the interleaving trace product of their composition T3 (if any) must satisfy the overall global security. For example, if member A is allowed to read a secure data, and A is also allowed to attach data to an email, the composition of the two functionalities might indicate that A is able to leak the secure data to untrusted sources. Components that satisfy one security property (eg, integrity) may be composed in such a way that the resulting system might not satisfy the property. In a situation like this, further security functionality/requirements will need to be added, such as logging all emails or merely preventing email clients from accessing the data. This example is a design vulnerability, in which the attack occurs by composing legitimate behaviours to produce an emergent abusive behaviour.

In order for the whole system to be secure, all relevant components must collaborate. When we put components together, predicting the consequences of their composition is difficult [194]. In the real world, a designer might explicitly specify that an application performs functions X, Y and Z. As these scenarios are composed together, a fourth scenario, W, might arise. Such an unspecified scenario, namely W, might result in a security failure if exploited by an attacker. Thus, it is important to discover all possible behaviours of the system, and to decide whether or not the behaviour is legitimate. It becomes more critical when we wish to support adaptivity and incremental development with respect to newly introduced functionalities, and the way in which these additions affect the overall composition of security. Thus, we need to have ways that allow us to incrementally and iteratively model, predict and evaluate the effects of composing behaviours together on overall system security. Executable architecture models support developers in exercising the integration of system components before they are built, in order to iteratively refine ambiguous requirements, search for hidden behaviour caused by the composition of func-

tionalities and generate test cases at early stages. These models also address implicit assumptions made by designers regarding the behaviour of system components. For example, a designer might assume that a particular piece of information is always present to the component; but in reality that information might not be present, which may lead to different behaviour of the component than originally intended.

When composed, requirements can contradict; thus, the role of the architecture manifests itself such that as requirements are elicited, it is possible to execute their composition and search for incompatibilities between requirements. A more holistic view of concurrent behaviour would be obtained by the merging of the partial views/representations in the behaviour, and would enable: (1) a better understanding of interactions across the composed design views; and (2) analyses of interactions to identify conflicts and undesirable emergent behaviours. If we are able to achieve this, then we would have the ability to create a composite trustworthy system using components as a heterogeneous collection of existing products.

3.6 Traceable and Observable

Previously, we have mentioned that security is not an 'externally observable property' [163]. Observability is defined as 'a measure for how well internal states of a system can be inferred by knowledge of its external outputs' [92]. This means that when we exercise the system under test, it is challenging to predict the internal security consequence by only observing the output (eg, error messages), because the output may only represent one of many behaviours that took place. In order to simplify the challenge, rather than attempting to observe the security property (eg, as in functional testing, see Section 2.4), we can instead observe the behaviour of the architecture under test as a composite artefact of the desired functionality. The need for security testing that supports observability is twofold: (1) either we refine the architecture to introduce new changes and we need to verify (retest) that the new changes did not violate any security requirement; and/or

(2) we need to verify that the overall architecture is secure and free of malicious and unintended behaviour. If the testing process is observable, we would be able to observe the extent of changes, and when they occur, in order to test their security implication on the architecture. As a result, we can verify that the overall architecture remains secure despite the changes that take place. In a situation where the security was compromised, having an observable security testing process would allow us to observe the internal state of the architecture to fully determine the security impact of the change and detect emergent behaviours. One way to achieve this goal is through observation of the dynamic composition behaviour of the architecture. Dynamic analysis of the architecture exposes the communication and control flow of the components, as it provides the set of possible traces of behaviour. Being at the trace level means that any changes would be visualised as either: (1) a newly emergent behaviour trace; (2) the absence of a behaviour trace; or (3) a change in the trace, and in return, we can focus the retesting process on the affected traces in the architecture. Not only will this provide observable changes, but we can also observe whether the desired functionalities (represented as traces of execution) are fulfilled by the architecture. This would allow us to trace back and forth between the requirement functionality and the architecture and to use these behaviour traces to confirm whether the implementation conforms to the architecture. In conclusion, the security testing process calls for continuous refinement, and refinement requires retesting because it may introduce vulnerabilities. As a result, the approach for security testing should be observable to provide assurance that changes triggered by refinement or modification of functionality can be identified and retested to maintain secure architecture.

3.7 Summary

We have highlighted the requirements for architecture-centric testing for security. The requirements motivate the need for an approach that is proactive in detecting design vulnerabilities at the architecture level. These vulnerabilities are design-specific; they arise

due to the insecure functional composition of the components constituting the architecture. These requirements call for an approach that supports incremental development and evolution for security. This is because the attacks landscape tends to change over time. Consequently, the architecture's security 'defensive' mechanisms shall be adaptable to the change. The requirements call for an approach that supports observability of changes that impact the security of the architecture, and promotes architecture refinements for security. The next chapter proposes an implied scenario approach for realising these requirements.

4

Implied Scenarios for Security Testing

In Chapter 3, we have highlighted the requirements for testing security at the software architecture-level. In this chapter, we pursue a test-driven architecture-centric approach to address these requirements. We describe a novel approach that exploits the concept of Implied Scenarios (ISs) [185] to test the security of system architectures. It introduces the concept of Security ISs to deal with vulnerability analysis at the architecture-level.

The approach aims at: (1) predicting the security behaviour of the architecture, its constituent interacting components and its relation to the operating environment, (2) systematically identifying areas of the system with potential vulnerabilities and consequently focusing the testing process on those areas, and (3) supporting the refinement process of the architecture for security in the presence of uncertainty and incomplete knowledge. These requirements aim to raise the level of abstraction at which security software testing is performed through leveraging on the architecture as a key artefact for promoting scalability and systematic guidance in the testing process.

The approach builds on Uchitel et al.'s (2003) IS detection for elaborating early requirements using scenario-based specification and behaviour models. We explore the fitness and effectiveness of IS in discovering vulnerable security states. Throughout this thesis, we show how ISs provide a basis for searching for insecure behaviour when testing the security of the software architecture so that it facilitates the building of security into the design.

Parts of this Chapter are published in [5] and [6].

4.1 Overview of ISs

4.1.1 Background

Scenarios define a temporal ordered sequence of events. The scenario-based design approach has been a popular technique for capturing the functional requirements of the user at the level of interactions between components [164, 105, 171]. This method provides a

concise yet simple tool for 'painting a picture' [59] of how actors, components and messages are composed together to complete one or more system goals. The messages and the sequences that pass between components in a process can be described using a message sequence chart (MSC) [84] or sequence diagrams in the unified modelling language (UML) [66]. In software architecture (SA), scenarios have been employed in modelling architectural properties [121, 167] and in identifying component interactions [121, 167]. Each scenario in an architecture models a partial description of desirable behaviour. Because these scenarios are partial, they cover most common system behaviours and the main exceptions. Expecting stakeholders to produce a set of scenarios that cover all possible system traces at once is unrealistic and impractical. For this reason, in the real world, we often encounter undesirable behaviours when partial behaviour scenarios are composed together. These undocumented behaviours (known as implied scenarios (ISs) [14]) may arise *when the desired global behaviour is implemented component-wise*; thus, components in concurrent systems only have local views of the execution where the order of scenarios is not explicitly enforced. They are 'implied' because they are not described in the specification.

4.1.2 Uses of ISs

An IS may be: (1) an acceptable (positive) scenario that has been overlooked and the scenario specification needs to be completed (eg, a new requirement may be identified); or, alternatively, may represent (2) an unacceptable (negative) behaviour in the system. Our interest in IS detection lies in that it searches *beyond the specifications* for behaviours that are not directly perceivable to testers (ie, unknown); thus, its presence indicates that we are not fully aware of the entirety of possible behaviours that may be exhibited. They allow us to capture uncertainties regarding communication amongst components. To claim that an application is secure, it is necessary to attain knowledge about the system's precise behaviour, and to confirm that all its behaviours are secure. Classic problems in software engineering, such as unexpected interactions between different system components, race

conditions and violations of hidden assumptions, are difficult to detect, even for expert testers. These vulnerabilities can be introduced by the functionality of the system that is not normally associated with the security requirements of the system. And since most of the attacks are a result of improper or unexpected interaction with the system [144], negative ISs help in identifying vulnerabilities that arise from unexpected interactions between (apparently correct) system behaviours and allow us to use this information to investigate the security status of the system.

4.1.3 Origin of IS

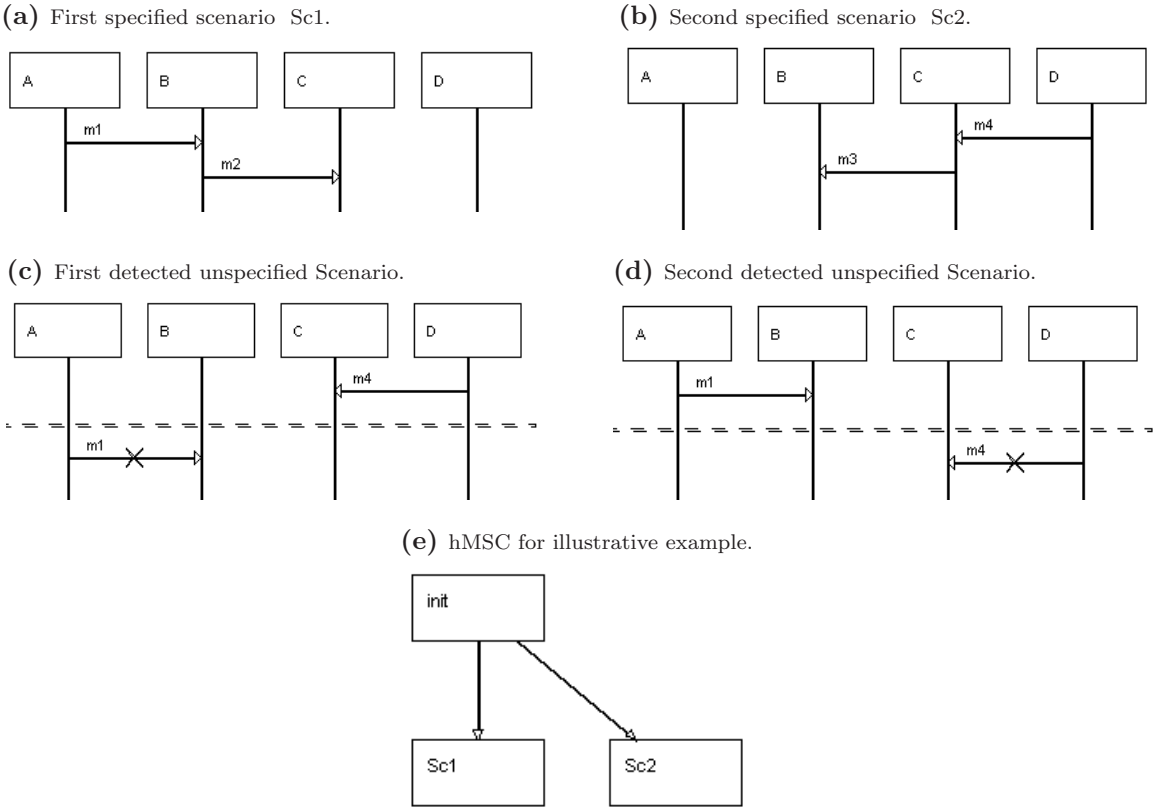
The notion of ISs was first introduced in [14] for a restricted scenario language. Their work is limited to a set of MSCs that specify a finite set of system behaviours. It was later extended by Uchitel et al. [186] to provide a more expressive scenario language that allows for an infinite number of system behaviours. The extension introduced an algorithm that analyses scenarios modelled in MSC specification, with a directed high-level MSC graph (hMSC) that defines the possible continuations and loops between scenarios (see Figure 4.0e). The hMSC provides the means for composing MSCs, where each node in the hMSC is a start symbol, an end symbol or a rectangle enclosing a reference to a MSC or another hMSC. Components are not shown in an hMSC since they have no meaning to the high-level sequence. The hMSC, with the mapping of nodes to MSCs, shows how the system can evolve from one scenario to another. MSCs are visual aids to design system specifications, yet their combined behaviour is still difficult to analyse through human observation. A behaviour model is a precise abstract description of the intended behaviour of a system in the form of a labelled transition system (LTS) [94]. It provides a view of how components interact. It structures systems as autonomous, concurrent entities and describes how they interact. These models often exist in complex systems because engineers construct them before building the software itself, as a cost-effective approach to correcting software design errors.

4.1.4 Applications of IS

We have done exhaustive research on the application of ISs in the literature, and have found that the main fields that have benefited from ISs are requirement elaboration [186], verification of web services [59] and reliability analysis [147]. In the field of requirement elaboration, ISs served in incrementally searching for new requirements. As an IS is detected, the stakeholder decides whether that behaviour is desirable or should be removed. In regards to reliability analysis, Rodrigues et al. [147] presented an automated approach for predicting software system reliability. They looked into the probability of component failure and scenario transition probabilities derived from an operational profile of the system. They also showed how ISs induced by the component structure and system behaviour described in the scenarios can be used to evolve the reliability prediction. On the other hand, Foster et al. [59] proposed an approach for verifying web service composition interactions for a coordinated service-oriented architecture in distributed systems. ISs allowed them to perform early verification of service implementations against design specifications, and to ensure compatible interfaces with respect to acceptable compositions. We have also seen the detection of IS at the execution level [167]. It gives insights as to how the application's intended behaviour is realised in terms of its implemented operations.

We have found that the concept of ISs as gaps in the specification fits very closely with the security testing purpose of finding hidden behaviour in the system. We have also found that the application of IS had not been used for security testing or for security analysis in general. As a result, we intend to use IS to benefit from its ability to predict design flaws/vulnerabilities in the architecture caused by improper compositions of functionality, and to promote architecture refinement to evolve into a more secure design. Our application of IS focuses on testing the resilience of architectures with respect to security.

Figure 4.1 An illustrative example of hMSC and MSCs of specified (a,b) and unspecified (c,d) scenarios.



4.1.5 Motivating Example

Suppose we have four communicating components in a system, for which we know that two scenarios are allowed to take place (as shown in Figures 4.0a and 4.0b); the execution of MSC scenarios Sc1 and Sc2 determines two possible and valid traces: $\langle m1, m2\rangle$ and $\langle m4, m3\rangle$. Each of these traces has a causal order, for which we know that (eg, in Sc1) message m1 should occur before m2. When the two scenarios are composed together (to model the occurrence of both behaviours at the same time), the interaction of these messages may not be anticipated, because the causal relation between the events of both scenarios is not specified. The example in Figures 4.0c and 4.0d reveals two additional possible executions outside the specified valid range. This is not surprising as the messages involve different components; and because there are no explicit synchronisation mechanisms to enforce particular ordering of these messages, any interleaving can occur. The

importance of this for security testing is significant because the understanding and approving all possible behaviours provides assurance that the system cannot be misused to act beyond the specified behaviour. This example demonstrates that changes in the scenarios will reflect directly on the ISs detected. In other words, different architecture models will yield different ISs, because the analysis is design-specific and, hence, the security status of each architecture model will be different. Notice that studying the specified scenarios (ie, the specifications) is not enough, and that exploration of the possible behaviours beyond the specifications is generally difficult for humans to perceive [188].

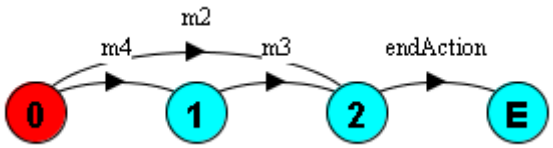
4.1.6 Detection of IS

We have briefly looked at what ISs are, and we have concluded that they arise because of lack of complete synchronisation across components. The extension provided in [186] introduced an algorithm that analyses scenarios modelled in MSC specifications. We have seen that these scenario MSCs constitute of message events sent and received between components, which are linked through a directed hMSC graph. The process of synthesising these MSC scenarios into Labelled Transition Systems (LTS) provides a way to computationally and mechanically analyse these scenarios to determine whether the behaviour specified is desirable given a complete system behaviour model. The LTS has been widely used for specifying and analysing distributed systems [34, 65]. The authors of the LTSA-MSC use a LTS to model each component of a MSC specification and use transition labels to model the message components send and receive. This allows for correcting, elaborating and refining scenario-based specifications through experimenting and the replaying of them [186]. They build a safety property that accepts traces that behave correctly according to the MSC and hMSC specifications, and checks that the synthesised LTS model satisfies such property using LTSA. If all component behaviour models are composed in parallel, it can be shown that the resulting system architecture can exhibit all the traces specified in the MSC. If a trace violates the safety property, then the trace is detected as IS. A formal syntax and semantics for MSCs is described in [186, 185].

Giving the hMSC graph with the scenarios in the example in Figure 4.1, the algorithm performs the following steps:-

1. Breaks down the continuation/loops of scenarios into several individual LTSs for each component. One component model is synthesised at a time, representing its behaviour across all scenarios. This is accomplished by (a) collecting the component behaviour in all MSCs, and (b) linking all its behaviour according to the hMSC. Each component behaviour model is synthesised to interact with its environment using only the messages that are allowed on its interface. For component C, we will have the LTS model shown in Figure 4.2, where messages $m3$ and $m4$ come from MSC Sc2, and $m2$ comes from MSC Sc1, and the component will either start executing message $m2$ or $m4$ in order to move to the next state. The sequence $\langle m4, m3, endAction \rangle$ is called a *trace*. A trace of a LTS is a sequence of observable labels that can be produced by executing the LTS.

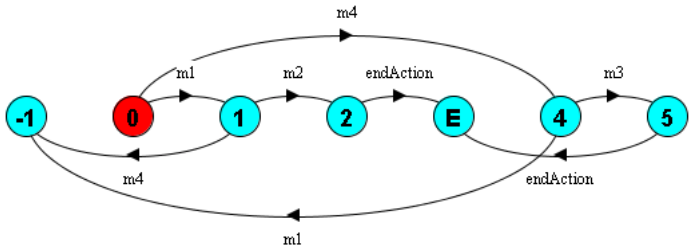
Figure 4.2 LTS model for component C



2. When all LTS models of the four components A, B, C and, D are individually synthesised, they are composed in parallel to one another in order to create the architecture LTS model of the system by (a) mapping nodes to behaviour and to their adjacent nodes, and (b) then connecting nodes according to the hMSC. This architecture model takes into account the behaviour described by the MSC scenarios (thus preserving the component structures and interfaces) and the global behaviour described by the hMSC. The joint behaviour is the result of all LTSs executing and sharing messages synchronously. Thus, the LTS can perform a transition independently of the other LTSs, as long as the transition label (ie, message) is not shared with the other LTSs [186]. Shared labels have to be performed simultaneously, thus components' LTSs that share a message will need to wait for each other to move simultaneously to the next state. Because non-shared transitions can

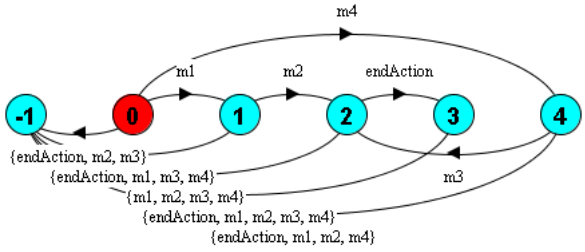
be performed independently across different LTSs, the notion of the architecture is weak because it allows for additional unspecified behaviours to emerge that are valid paths in the hMSC, and which exhibit all traces specified by the MSCs. The architecture model produced in this synthesis provides a complete view of the system design, allowing for the concise inspection of all possible paths and behaviours of the composition. The complete architecture model of our example is depicted in Figure 4.3.

Figure 4.3 Architecture model for the illustrative example presented in Figure 4.1



3. Builds a new trace model that captures the exact set of traces defined by the MSC semantics (ie, the required behaviour) to compare it to the behaviour of the architecture model. Any additional traces in the architecture model that are not specified in the trace model correspond to ISs. This trace model (as shown in Figure 4.4) is built using a coordinator component that is responsible for allowing/disallowing components to move from one MSC to another. This way, components can be guaranteed to not follow different sequences of MSCs.

Figure 4.4 Trace model shows the precise desired behaviour of the illustrative example in Figure 4.1



4. Compares whether the architecture model exhibits more traces than the trace model. This is done by (a) defining the trace model as a safety property that accepts traces that behave correctly according to the MSC and hMSC specifications; and (b)

checking whether the architecture violates the property. Violations are then reported as IS MSCs. When we compare the architecture model in Figure 4.3 to the trace model in Figure 4.4, we can see that the architecture model exhibits two additional traces that are not specified in the trace model. These two traces are depicted in Figures 4.0c and 4.0d

A formal definition of ISs is available in [186].

4.2 IS for Security Testing

In the previous sections, we have stressed the importance of understanding the system behaviour beyond the specification. Emergent behaviour may have serious security consequences on the system, and their detection is important to ensure that the system always behaves as intended. As many early requirements cannot be recognised at once, IS sets the path for evolution opportunities to build secure systems, because future changes are generally unanticipated [21]. In the past, we have learned that:

- Attackers intentionally probe unspecified behaviours in the system. They attempt to make the application behave in an unanticipated manner, and then determine the attacks associated with that behaviour (see Section 3.3).
- Security testing is motivated by addressing undocumented behaviour and areas of particular complexity to determine how a program can be broken [80] (see Section 2.1).

We have reviewed the requirements that need to be realised when addressing security testing at the early phases of development in Chapter 3. In this section, we will reference these requirements and explain why the concept of ISs serves to achieve them.

1. The existence of ISs indicates unexpected, yet possible, system behaviour. They are precisely what the application can do beyond the specification. It gives clues to

insecure behaviour that is outside the range of the legitimate usage. Its predictability of potential vulnerabilities allows us to proactively and systematically discover weaknesses in the architectural decisions we are about to make. Correcting these problems at early stages prevents them from manifesting into the design and implementation of the system and, consequently, we are no longer in a reactive/defensive position. We are now able to predict/anticipate attack moves and secure a design before vulnerabilities are exploited. This is important because testers are required to think like attackers and search for ways to abuse the system to make it behave incorrectly.

2. ISs are behaviours that arise from the design itself (ie, they are design-specific artefacts). The discovery of ISs is a means to exploit weaknesses in the design without attempting to re-run predefined test cases of other applications (which serves very little in securing the design). The concept of ISs builds on the incremental and iterative nature of development. It permits a systematic approach for handling changes to enhance the security of the software. When a refinement is necessary (eg, a new requirement is added), we are able to observe the consequence of the addition to the rest of the system before we commit to the refinement. This is done by observing the appearance of new ISs after the refinements are made. If security related ISs are detected, the testers can determine if a refinement is advisable to maintain the security of the architecture. This test-driven approach enforces the concept of Security By Design, because every step taken contributes to the overall security of the design. It also offers a systematic approach for handling changes.

3. ISs are the consequence of functional composition. They serve predicting the dynamic behaviour components at an abstract level. Given a set of specifications, we are able to verify that all relevant components collaborate to ensure that the desired global security is achieved. They allow us to study the security level of compositions at the control-flow level, after which we can determine whether or not security is

preserved under certain compositions. It is a powerful mean for addressing the uncertainty of compositions, especially when many big systems are built using smaller components gathered from different sources. Given an IS execution trace, testers can visualise how these scenarios can be triggered by attackers, and how the execution of these scenarios can effect the security of the system. Furthermore, ISs drive the focus of security testing on the vulnerable component interactions, thus enabling the assessment of third-party-components by observation, even when the code is not present. Understanding the control-flow of communication increases the likelihood that the design transferred to production will translate into a trusted implementation to reflect the refined architecture. Consequently, we can confirm whether the trusted architecture design is implemented in practice when the application is developed, as one can verify using a set of test cases, whether the traces are implemented in the system as desired.

4. Functional scenarios used to model the architecture represent desired requirements. As a result, detecting positive ISs can be used to verify whether the architecture continuous to meet the desired functionality as the architecture evolves.

5. Even though some security experts believe that architecture-centric flaws can currently be found only through human analysis [12], the nature of ISs leads to difficulties in manual detection. We thus benefit from semi-automation, which reduces the time required to search for these scenarios. Unlike fully human-centric approaches, automation is less prone to errors and the overlooking of threats.

4.3 Architecture-Centric Testing for Security

We advocate an architecture-centric approach for security testing using IS detection. We adopt a testing approach that provides predictability with respect to likely composition behaviours as changes continue to evolve. We aim to testify the security of composition

with respect to breaches of confidentiality, integrity and availability (CIA) properties. This provides an insight into the security status of the architecture. The questions we explore in this thesis are: (1) How important is the detection of ISs for security? (2) What consequences can ISs have on the security of the system? and (3) How do we benefit from this information to build secure systems? To begin with, this thesis builds on the following assumptions:

- The specified functional behaviours will naturally be legitimate and desirable (ie, secure) as they are the direct result of carefully considered requirement and architecture phases.
- We can evolve a secure-by-design architecture as changes are applied to the architecture by advocating a test-driven process to verify the security of the architecture.
- The software architecture is secure if and only if, it behaves as intended by the specification.

The architecture models guide the comprehensive development of security. To build a complex system, it is difficult for one to account for every aspect of the system design. The state-of-the-art approach to managing complex systems is to adopt architecture models [47] for systems to simplify the description and provide a holistic system view. Such high-level modelling enables designers to locate potential vulnerabilities and install appropriate countermeasures early in the development. The system architecture we use in our approach consists of the following information:

1. The *component structure*: which consists of a list of all the components that appear in the modelled scenarios. These represent the computational elements and data stores of a system, and communicate with each other through message passing.
2. The *component interface*: which is the means of interactions among components; given a component C of some scenario, the interface of C is determined by the set of messages that are sent and received by this component.

3. The *configurations*: which represent the topology arrangement of components that sets the rules for composing components together.

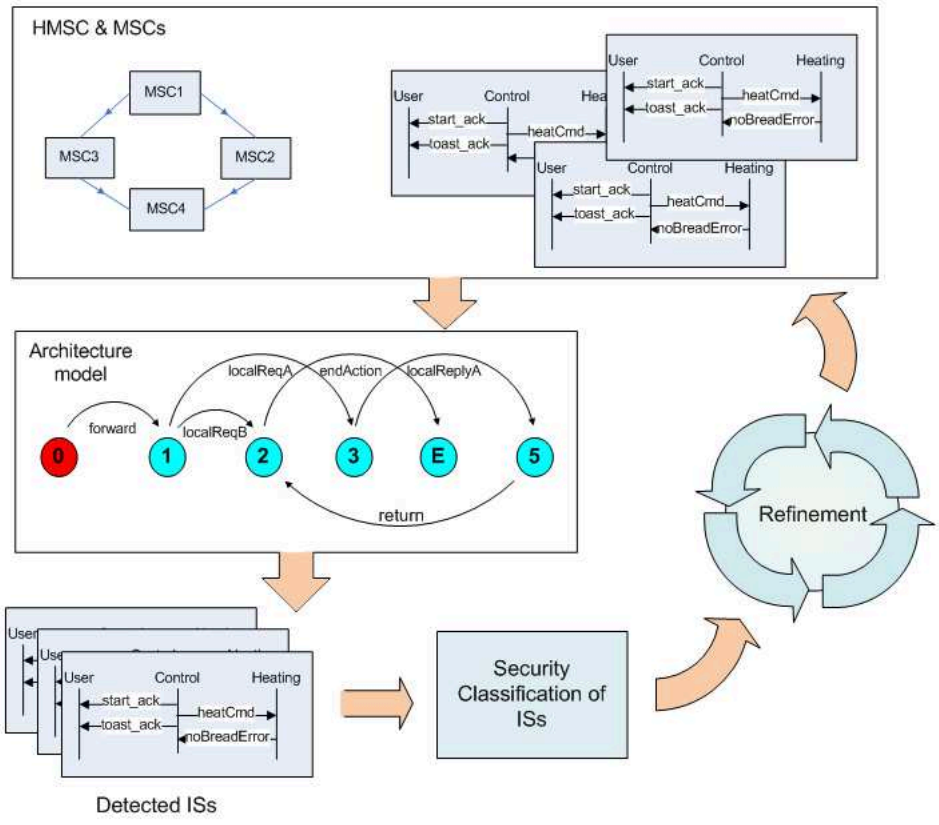
Security experts consider the architecture and design to be the single most critical phase of the secure design life cycle [12, 163, 113], because good decisions made during this phase yield an approach and structure that are resilient and resistant to attacks. The architecture provides sufficient levels of abstraction (such as message passing and interfaces) that hide the unnecessary details of complex code-level interactions. We recall that security testing is the process of identifying vulnerable behaviours that affect the CIA properties of any parts of the system (Section 2.1). At the architecture level, security testing is the process of detecting emergent behaviour that diverts from the intended behaviour and affects the CIA properties [6]. Our objective is to test whether the architecture remains secure as we compose new information. This testing process involves: (1) identifying malicious behaviours; and (2) identifying their security implications on the architecture (ie, the negative result of their correct execution, such as breach of confidentiality). ISs are precisely these behaviours that must be detected and tested for their security conformance/implications.

ISs have been used as a way to elaborate requirements in an incremental fashion. We extend the definition of IS to contribute to the field of security testing by viewing ISs as unanticipated (possibly malicious) behaviours that indicate potential insecurities in the architecture. Our approach adopts ISs as a mean to reveal security vulnerabilities that guide the process of architecture refinement. These ISs are indicative measures of how secure the architecture is and how many potential weaknesses need to be evaluated in the specification. They can be used to predict the security of architecture refinement. If, for example, a change is insignificant with respect to functionality, but can result in new ISs that impacts the security, then we can determine whether a change is advisable or should be withdrawn. The fewer hidden IS behaviours, the better security we have, because this provides assurance that the architecture behaves as intended. Furthermore, the testing process involves assessing the security of the architecture with respect to presence of

unwanted (possibly malicious) behaviours. This means, if a refinement is expensive to be adopted, one would still want to analyse the extent to which that malicious behaviour can impact the architecture (eg, what resulting behaviour may branch from it). As such, it maybe possible to prevent some of its subsequent consequences. Eg, we maybe able to synchronise two components to prevent an unauthorised access.

4.3.1 Phases of IS approach for Security Testing

Figure 4.5 IS Approach: Takes in the MSC scenarios with the hMSC graph, then builds the architecture model to search for ISs. All ISs that lead to security breaches are classified, and are used to refine the architecture.



We propose an architecture-centric approach for security testing consisting of three-phases for security testing (as shown in Figure 4.5): (1) Detecting design-level vulnerabilities in the architecture in an incremental manner. (2) Classifying the impact of detected ISs on the security of the architecture. (3) Use the detected ISs and their impact to guide the refinement of the architecture. The refinement process is test-driven and incremental,

where refinement cycles are tested before they are committed. The method provides a proactive and early feedback on the security of the changes applied to the architecture, such that testers are able to make informed decisions about the refinements. Below we describe these stages in details:

Stage 1: IS Detection

Inputs: System specification in MSC specification + hMSC graph (see Figure 4.1).

Outputs: Detected ISs.

We assume that the system specification is presented in a scenario-based specification language describing the required functional behaviour, as well as the desired loop/continuation graph between the scenarios. These scenarios may reflect the desired specification model, or the implementation details of the code. The distinctions between the two models include:

- the specification model is likely to be present during the development, and thus no extra effort is needed to apply our approach, but it may not truly reflect the implemented system; hence, generated ISs may be false positives.
- the implementation may require waiting until the system is built, which delays the testing process. However, ISs from the implementation model are likely to reflect real vulnerabilities in the system.

The flexibility of the approach in adopting both situations is beneficial, because it can be used to test systems in their initial design, or to test previously built systems. Once the scenario specifications and the hMSC are present, we synthesise the scenarios using the LTSA-MSC tool to create the architecture model. We collect all the detected ISs that maybe: (1) design vulnerabilities, which are used to guide the architecture refinement, or (2) positive ISs that have been overlooked, which can be used to complete the design.

These positive scenarios can also be used to check the conformance of the architecture with the requirements.

Stage 2: Review of detected ISs

Inputs: Detected ISs.

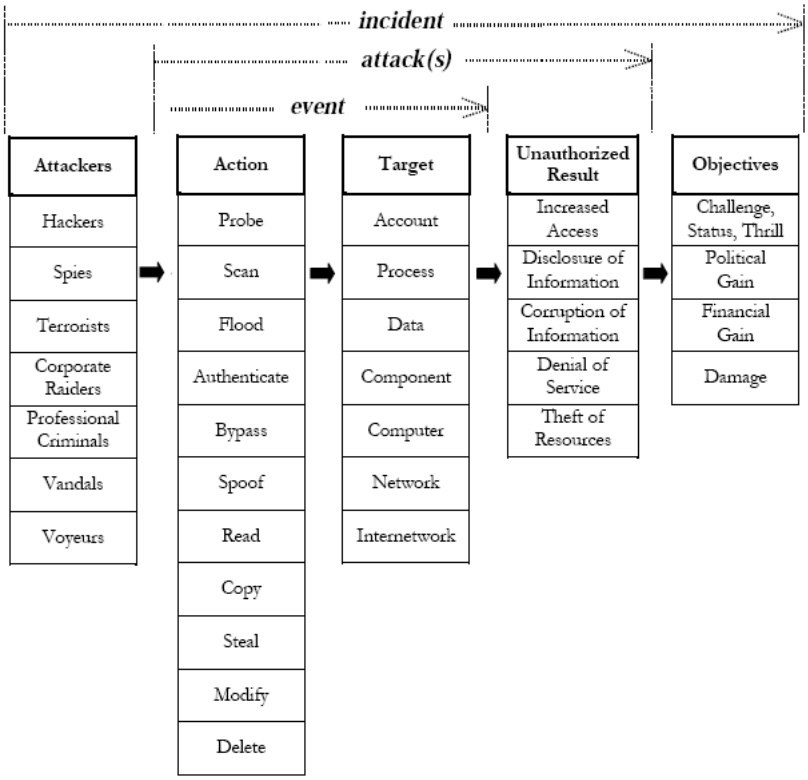
Outputs: Classification of detected vulnerabilities.

The focus of this step is investigating the potential security consequences of the detected ISs. An IS may cause several threats to the system, or maybe exploited in several ways. We use a widely used classification scheme [78] to realise the impact of ISs on the security. The classification scheme [78] as shown in Figure 4.6 looks at the types of attackers, their objective, actions, target and the unauthorised results of the threat. Given an IS trace, the testers will need to answer the following questions:

1. What negative security impact can the IS behaviour cause? Eg, Can detected IS cause 'information corruption' if triggered by an attacker?
2. Which of the listed attackers may have interest in executing the IS?
3. What would be the attacker's target resource behind the threat?
4. What would be the attacker's objective behind the threat?

The outcome of this process is a classification of the ISs containing: (1) the type of attacker that may launch the threat, (2) the description of the actions required to launch the threat, (3) the security breach results and (4) the objective of the threat. The classification is used to support testers in identifying the various ways in which a threat may manifest itself into the system, and the range of attackers who may have interest in exploiting the IS. It provides a checklist-based approach whereby testers can have reasonable confidence that all potential attack actions and results have been considered. Using the taxonomy, we can identify the types of threats detectable using ISs, and which architectures are more prone to certain threats.

Figure 4.6 Computer and Network Incident Taxonomy [78]



Stage 3: Refining the Architecture

Inputs: Current architecture.

Outputs: Refined architecture for security.

The aim of the refinement process is to correct the architecture model with respect to the Security ISs detected in Stage 2. The refinement may involve adding/removing functionality, or modifying an existing functionality to prevent the appearance of the Security IS. These changes to the architecture may lead to new ISs. The testers have a choice to deal with new ISs: either they roll back to the original architecture and consider other options, or they commit to the refinement and address the arising ISs individually. We say individually, because it allows us to identify the cause of the new IS, such that if we choose to withdraw we know precisely which refinement we need to roll back. This early feedback on the refinement informs the testers about the security of the refinements they intend to pursue.

The refinement decisions are design-specific, and rely on human expertise. Human involvement is needed primarily to evaluate the security of the detected ISs and to perform the refinement. The process of detecting ISs is relegated to a mechanical procedure in Stage 1, because it is difficult for humans to reason about interactions among system components. Although it may be possible to define security properties to fully automate the approach, we believe that such a step is likely to neglect important threats that do not violate the defined security properties. We have previously reviewed in Chapter 2 that fully automated tools suffer to address design-vulnerabilities.

4.4 Summary

We have presented an architectural-centric testing for security. We proposed the concept of Security IS, which is an unanticipated (possibly malicious) behaviours that indicates potential insecurities in the architecture. We proposed a three-phased method for security testing: (1) Detecting design-level vulnerabilities in the architecture in an incremental manner. (2) Classifying the impact of detected ISs on the security of the architecture. (3) Refine the architecture by using the detected ISs as a guide. The refinement process is test-driven and incremental, where refinement cycles are tested before they are committed.

In Chapter 6, we will demonstrate the applicability and effectiveness of the approach using three case studies. The application demonstrates novelty in exploiting the approach for security testing in dynamic and unpredictable environment such as cloud, distributed smart cameras and web applications. Drawing on these cases, the Chapter will report on the evaluation of the approach.

5

SecArch: Architecture-Centric Testing for Security

5.1 Introduction

In Chapter 4, we discussed the importance of ISs in detecting security vulnerabilities; we discussed the negative effects of inconsistencies and hidden interactions in regards to the security of software from a single dynamic behaviour view. In this chapter, we present SecArch, an enhancement to the IS approach to improve its search-space for threats. The extension is concerned with predicting more valid conditions in the face of real parallelism in distributed systems with respect to non-FIFO queues. It uses detected race conditions to test for security of the architecture with the presence of negative behaviour.

We begin by drawing a comparison between complete and incomplete models using two existing researches, one for detecting ISs [185] using behaviour models, and one for detecting race conditions using scenario diagrams [15]. We then explore the benefits of merging both views to detect ISs as well as race conditions. We propose moving from purely dynamic behaviour models (LTS models) to structural MSC models to preserve structural properties that are used to detect race conditions. The work reported in this Chapter has been published in [7].

5.1.1 Motivation

Research in analysing dependability with respect to interactions across multiple views has been limited/lacking [60]. Incorporating multiple views for security analysis allows for the understanding of possible consequences of negative behaviour across other views. In return, this can serve in detecting multi-step attacks, accurately estimating the associated risks with it or in making informed architecture refinements with respect to potential damage caused by the negative behaviour. Connections between behavioural and structural models provide extended means to analyse the security of concurrent systems. Incremental behaviour models target the incompleteness of scenarios by searching for gaps in the specifications and incrementally adding missing scenarios. These gaps might be due to an inconsistency of scenarios in which additional scenarios might be introduced, or legitimate

scenarios being overlooked during the requirement/design phases.

In this section, we investigate (1) the difference between structural and behavioural models and why we need both; (2) the importance of detecting race conditions from behaviour traces (rather than structural traces).

5.1.1.1 Behaviour vs Structural models

Behaviour models are successful in uncovering the subtle errors that can appear when designing concurrent and distributed systems [187]. They provide feedback on the accuracy of models. Certain types of dynamic behaviour models, such as LTSs and state charts, focus on the collaborating entities and study the different sequences of exhibited interactions. An example of an inconsistency could be an IS [14]; these scenarios arise due to limitations in the local view of each component involved in the interactions and do not require the syntactic checking of scenarios. Dynamic architecture-based analysis is concerned with demonstrating software-predicted run-time behaviour. Early analysis allows for the early detection of defects before the system is built. It is a necessary part of testing as it supports analysis with respect to operation environment variables. However, dynamic analysis on its own is insufficient to verify all architecture-concurrent defects, which require a thorough analysis of executions. Behaviour models can be rich on their own if: 1) the initial scenarios are close to completion; or 2) the synthesis algorithm is not constrained by a set of semantics that reduces its search space.

On the other hand, whenever concurrency is present, race conditions are possible. Checking for race conditions requires the preservation of the precise ordering of messages across all scenarios, which may not be preserved by the behaviour model. Sequence diagrams focus on the order in which messages are sent, and the procedural flow through components. This is done by individually checking the syntactic correctness of scenarios. However, syntactic correctness is not enough when dealing with concurrency, because analysing each sequence diagram for race conditions may result in: (1) only sub-traces being addressed rather than overall maximal execution; (2) the inspection of previously-

handled race conditions that are repeated in several interaction models; and (3) what might be reported as a race condition might be acceptable in another scenario in the specification.

Both behaviour models and interaction models are especially important in understanding the run-time behaviour of components. Generated behaviour models allow the tester to examine the dynamic behaviour from a different angle as compared to interaction diagrams. Each of these models focuses on certain aspects of interactions; thus, integrating information gathered from multiple views provides a multi-dimensional representation of the pre-developed system that features fewer design inconsistencies that may impact the dependability of the system. While interaction diagrams show the interaction between several objects, behaviour models show the total behaviour of all components in the system over time. Behaviour models can be used to rigorously analyse how a set of components, which comply with the architecture described in the scenario-based specification, behave when working together. In particular, we can analyse to see if they exhibit any traces that have not been explicitly specified in the scenario-based specification.

We summarise that concurrent systems require an analysis of both semantics implications and structural information of specifications. These representations are crucial to offset the limitations in both [41, 185]. We can also obtain a more holistic view of concurrent behaviour by merging the partial scenarios in the behaviour models and interaction models. Without incorporating multiple views, a false sense of completeness might result in a system deployed with untested hidden behaviours. In particular, we emphasized the need for incremental models when dealing with scenario-based specifications, as they take into account the nature of the development cycle and the continuous requirement for elicitation.

5.1.1.2 Model Traces

A trace of an LTS is a sequence of observable labels that can be produced through execution of the LTS. It models all possible executions of components from a global view of the

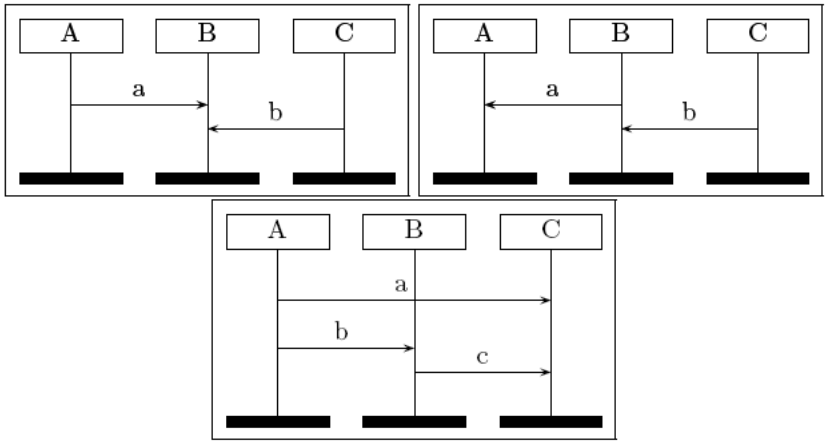
system (ie, the composition of scenarios). The traces we are interested in are maximal traces; these are the executions that cannot be extended any further. The architecture model in [185] (also in LTS form) features infinite execution traces of all possible behaviours of the components. On the other hand, we have the traces extracted from each individual MSC scenario specification, which represent the causal order of messages in that scenario. Fundamentally, LTS traces differ from the sub-traces extracted from each individual MSC scenario because they do not model the scenarios individually; instead they model: (1) the composition of scenarios from multiple component views; (2) the possible continuations of a scenario; and (3) hidden ISs. However, as the LTS model is a set of reachable states that describes how a system corresponds to events, it is not possible to use the LTS model to search for race conditions unless further annotations of message ordering and component names are preserved. Instead, scenario-based specifications, such as [84], focus on the procedural flow through components. Their structural properties play a vital role in detecting race conditions in concurrent systems.

5.1.1.3 ISs vs Race conditions

ISs are different in nature to race conditions, even though both are emergent behaviours. Race conditions are among the most common form of inconsistencies [128, 57, 49, 15]. Essentially, a race condition asserts that a particular order of events will occur as a consequence of the causal ordering (ie, visual order), when in practise this order cannot be guaranteed to occur due to limited control of the speed of the message propagation in practice. *A race condition exists when two events appear in one (visual) order in the MSC, but can be shown to occur in the opposite order during an actual system execution* [15]. Race conditions occur when different processes access shared resource without explicit synchronization [130]. There is often no way to ensure that two messages from different processes arrive in the same order. This means that the visual order may provide more ordering over events than is achievable in practice. Figure 5.1 shows three classes of race conditions in an MSC scenario [49] where the order of receiving messages A, B and C may

not be maintained. These classes of races are limited to the trace being investigated (i.e. lack of order of a specified scenario trace). It is very easy to inadvertently introduce race conditions into a scenario, because causal orders semantics places almost no constraint on how the causal order is constructed [120]. On the other hand, we have seen that ISs arise due to lack of information in the component view, and that their occurrence is outside the specified set of traces due to dynamic compositional fault. The commonality between both race conditions are ISs is in the lack of synchronisation between components. We cannot use a detection algorithm dedicated to finding ISs in order to find race conditions, and vice versa.

Figure 5.1 Three basic types of race condition [120]



5.1.1.4 Illustrative Example

We have mentioned previously that the IS detection algorithm [185] uses hMSC graphs to determine the right path to traverse the scenarios. A shortcoming in this detection algorithm is that when there is a branch in an hMSC node (ie, two parallel scenarios), the synthesis algorithm combines the two branches/scenarios using an 'OR' logical operator rather than an 'AND' logical operator, though it synchronises on shared messages. This allows a race condition to be bypassed when two components in both branches are trying to access a shared resource. Their event semantics are built on strong assumptions about the ordering of events, such as only supporting single-queues (ie, enforcing visual order).

For a race condition to be detected, only causal orders should be preserved. Restrictions on queues make for unrealistic assumptions about the components involved in the concurrent system. Alur et al. [14] have presented an analysis tool that can perform automatic checks on message sequence charts with respect to timing constraints of the message-queuing structure (such as FIFO) and possible semantic interpretations, and can detect conflicts such as causality cycles and race conditions. We use Alur's algorithm because it allows us to check for race conditions based on several queue structures, filling the gap in IS detection algorithm.

Suppose we have a scenario in which two users (client and admin) are trying to access the same server; an admin can enable/disable the server, while the client is using the server. Both users can perform operations on the server at the same time. However, when we compose the individual LTS models of each component using an 'OR' logical operator, then we are assuming that the component has a single-FIFO queue, and can only deal with one request at a time. This does not reflect true parallelism. The traces associated with each individual scenarios (see Figure 5.2) are as follows:

```
1: enableServer
2: disableServer
3: login > successful > AllItems > selectItems > returnItems> buy > logout
```

Searching for race conditions on each of the above traces does not detect race conditions. On the other hand, a behaviour model (see Figure 5.3) would allow us to merge the occurrence of all these scenarios/traces with respect to the hMSC. *Using traces in such models to generate scenarios reveals another side of the communication unperceived in the initial scenarios.* The traces from the behaviour model are as follows:

```
1: enableServer > disableServer
2: enableServer > login > successful > AllItems
    > selectItems > returnItems> buy > logout
    > disableServer
```

Figure 5.2 Three Client/server scenarios with their HMSC

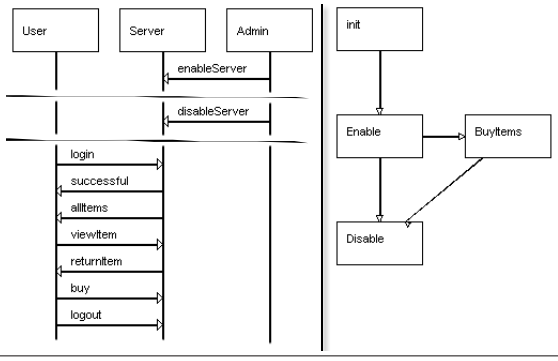
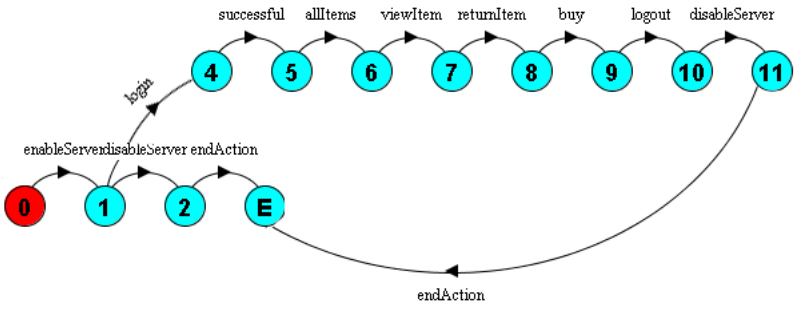
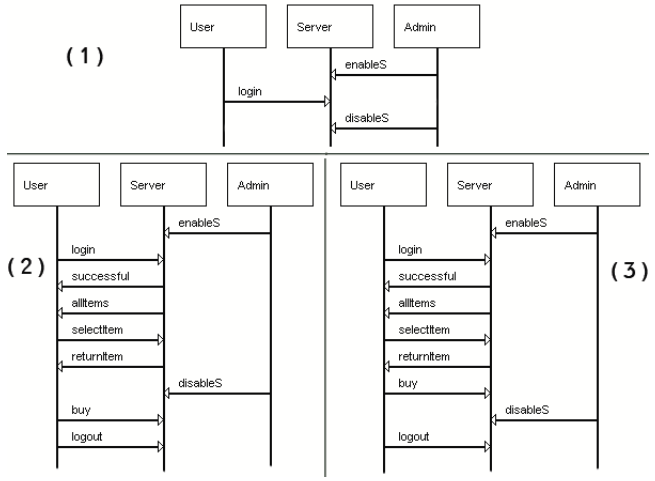


Figure 5.3 Behaviour model for the Client/server example presented in Section 5.1.1.4



However, the real number of possible interleavings exceeds the above two traces. There are 10 more possible traces that can be exhibited by the above scenarios (examples can be seen in Figure 5.4). This is due to the assumption of single FIFO queues in a synchronous communication, which leads to the isolation of the scenarios from the server's view (ie, the server contains a single queue that deals with one request at a time). But in a realistic setting, concurrent systems are likely to have non-FIFO queues, or FIFO queues with asynchronous communication. We will revisit this example in Section 5.2.1.

Figure 5.4 Detected Race Conditions found in the client/server example



5.2 SecArch: Architecture-Centric Testing for Security

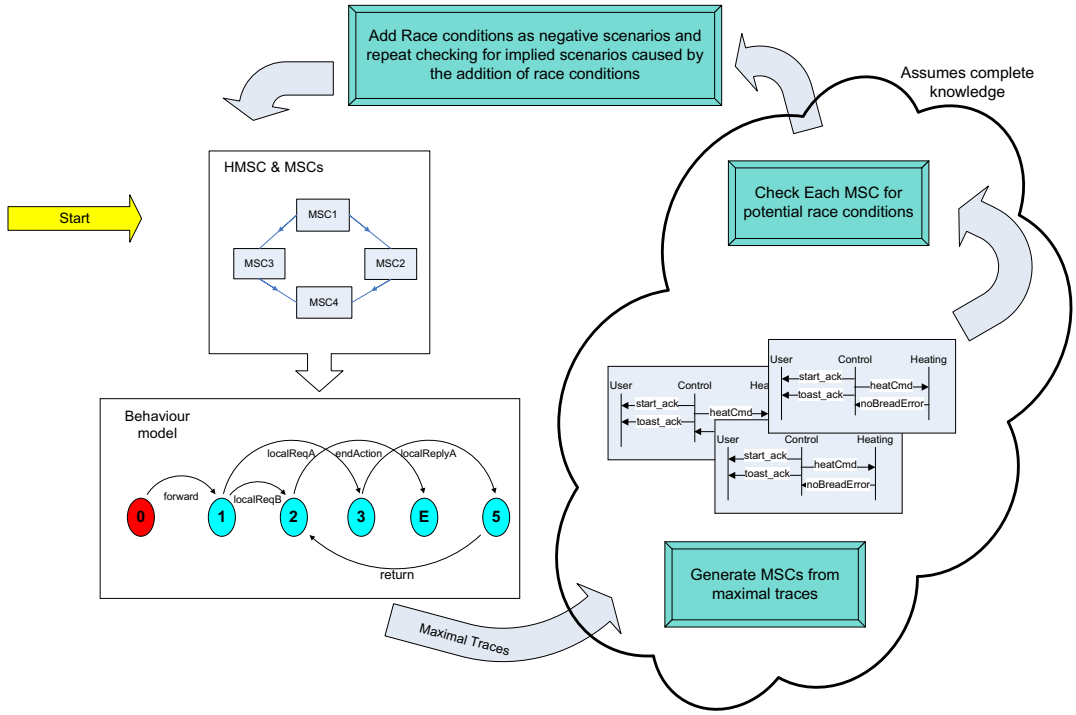
To examine the potential benefits of combining IS detection with race condition detection to reveal insecure behaviour, we introduce the use of Alur's algorithm [15] for detecting race conditions, in conjunction with the IS detection we presented in Chapter 4. Our SecArch is intended to complement the limitations in both algorithms in order to detect further insecure behaviours by preserving the dynamic and structural aspects of both algorithms. Table 5.1 shows the feature-comparisons of both algorithms as well as our SecArch proposal.

Table 5.1: Attribute comparison between Uchitel's algorithm [185], and Alur's [15], and SecArch

Criterion	Uchitel	Alur	SecArch
Assumes complete knowledge of environment		✔	
Assumes incomplete knowledge of environment	✔		✔
Produces all possible executions of modelled system	✔		✔
Multi-scenario analysis (i.e. produces maximal traces)	✔		✔
Single scenario analysis (i.e. produces sub-traces)		✔	
Searches for specification gaps (i.e. ISs)	✔		✔
Searches for race conditions		✔	✔
Supports High-level MSC to infinite traces	✔		✔
Finite traces (i.e. bounded MSCs)		✔	✔
Syntactic analysis		✔	✔
Semantics analysis	✔		✔
Models timing		✔	✔

To address the limitation discussed in Section 5.1.1, we propose an additional stage to stages 1 and 2 introduced in Chapter 4 (section 4.3.1), in which we take behaviour model traces for structural analysis using Alur's algorithm for detecting race conditions. Once negative race conditions are detected, we reimburse Uchitel's algorithm with the new traces iteratively, and repeat the process until no more ISs or race conditions are detected. This incremental search for race conditions ensures that complex race conditions are found with every newly added functionality. Figure 5.5 presents all the steps required in SecArch approach.

Figure 5.5 SecArch: takes in MSC diagrams, then builds a behaviour model to search for ISs, and finally degenerates MSC diagrams from the behaviour model traces to evaluate for race conditions, and replays dangerous race conditions back into the model for further ISs evaluation.



Stage 3: Race condition detection

Inputs: Architecture traces.

Outputs: Refined architecture for security.

To perform this step we need the list of traces in the architecture. To extract the traces, we used a built-in simulator in the LTSA-MSC. These traces are modelled in MSC to check their structure for race conditions. We use the UBET [102] tool to search for race conditions which implements the Alur's algorithm. When a race condition is found, we review its security impact using Stage 2 of the IS approach. If negative race conditions are found, we feed them back into the LTSA-MSC tool as positive IS and update the HMSC, then we repeat the process. Our use of ISs in this proposal is different in comparison with the literature discussed in section 4.1.4; we do not classify bad scenarios as 'negative' to constrain the model. Instead, we accept all ISs as positive scenarios to examine how the system reacts in the presence of bad behaviour. This step is crucial for two reasons: 1) for security testing, two-step attacks rely on the presence of a weakness in the design (that might not necessarily cause a problem on its own) to launch a more sophisticated attack; and 2) it is often expensive to redesign a system once it is built; thus, an approach is necessary that adapts to the presence of weaknesses and provides insight into how such weaknesses can affect the system.

5.2.1 Example Application of SecArch

Referring back to the example we introduced in Section 5.1.1.4, applying SecArch leads to the following race conditions (see Figure 5.4) with respect to security testing:

Race Condition 1: Disabling the server during authentication. The interest is in realising how the system behaves when an admin interrupts the authentication service. Does it return user data back in the error message (which can lead to a reflected XSS attack) [189]? Does it dispose of credentials securely or does the data remain floating in the server's memory? Does it return a stack-trace or code back to the user that reveals

a great deal of information about the server structure and implementation? The answers to these questions may reveal further critical threats.

Race Condition 2: The user commits to buying an item while the admin is disabling the server. We are mainly concerned with the application's protection of the confidentiality of the data during the termination. For example, the payment might be rejected gracefully or disrupted during processing. Consequences can range from money withdrawals from a bank account without being recorded in the database, or bank details and private information remaining in the server's memory without secure disposal or possibly, a double processing of purchases.

Race Condition 3: Disabling the server while the user is logging off. Will the user remain logged in once the server is enabled? This would pose an impersonation problem if someone else uses the account.

All the above conditions may lead to problems in the deployment time –depending on how the server operates– if not tested properly and carefully, either in terms of sessions, error handling or memory garbage collection. At the architecture-level, the testers may introduce new functionalities that deals with each specific case. For example, the disable function maybe designed to confirm that there is no critical process running before terminating. This guided systematic testing helps to highlight potentially vulnerable areas, to reduce the possibility of overlooking vulnerabilities and to reduce searching time when manually debugging. As is clearly shown, both Alur's and Uchite's approaches individually failed to detect these scenarios, but the combination of both complements the limitations and proves to be useful when considering potential problems in a systematic fashion. Although SecArch assumes incomplete specifications to support the partial nature of the scenarios, we are benefiting from Alur's assumption of complete specification to reimburse the specifications with further hidden behaviours. The evaluation continues to run in cycles until no more ISs or race conditions are found.

5.3 Summary

In this chapter, we presented SecArch, an incremental architecture testing method that merges behaviour models with structural analysis for improving the search-space for threats and the detection of inconsistencies. We merged the concept of IS [185] detection with race condition detection [15] to complement the limitations in both, and discussed how the integration of the two approaches is more effective in guiding testers to detect potential security threats than either would be individually.

In Chapter 6, we will look at novel applications of SecArch for security testing in dynamic and unpredictable environment such as cloud, distributed smart cameras and web applications. We will evaluate its efficiency, its scalability and applicability.

6

Case Studies and Evaluation

6.1 Introduction

In this chapter, we will evaluate the approach through reflection on its ability to meet the requirements discussed in Chapter 3. We will study its applications to address timely and challenging cases characterised with distribution, parallelism, dynamism, unpredictability of operation and openness. Though the primary goal has been to explore the fitness of the approach for targeting security testing at the architecture level, these applications shows novel perspective in how IS and SecArch can be deployed to address problems in emerging domains such as the cloud. We argue that our approach is suitable for security testing as it enables testers to look beyond the specifications for hidden and malicious behaviours, and to use that information to systematically guide the architecture refinement process.

Evaluation of the thesis aims at extending the confidence in the following claims:

- ISs are design-specific behaviours that emerge unexpectedly from functional composition. We want to verify their security impact and to increase confidence in the ability of the system to withstand these breaches.
- Our approach has the ability to systematically identify critical deviations from the intended behaviour.
- IS approach can guide the architecture refinement process to reach a secure architecture. The refinement can revisit the architecture design decisions and incorporate architecture mechanisms/tactics, which can prevent the likely impact of IS emerging behaviours.

The evaluation aims to testify these claims through illustrative examples drawn from the case studies. We reflect on the strengths and weaknesses of the approach.

6.2 Case Study Setup

In this section, we look at the tools used, the method's protocol, research questions, propositions, and data collection.

6.2.1 Software and Hardware

The approach makes a novel use of a number of sound tools to assist the process of IS detection, classification and architecture refinement for architecture-centric testing for security. The architecture is realised through composition of functional scenarios (which reflect the desired functional requirements) using the LTSA-MSC tool [185]. This tool is also used to generate ISs. It takes the scenarios in MSC specification and the hMSC graph that models the continuous flow of the scenarios (ie, how the scenarios are composed together) as an input. Another tool is used to detect race conditions, UBET [102], implements Alur's algorithm to detect race conditions [15]. These tools have been used in a number of publications (eg, LTSA-MSC was used in [187, 147, 59], UBET was used in [150, 22, 24]). However, they were not previously used to exploit the benefit of Security IS detection and architecture refinement. All case studies were run using the Windows XP operating system with an Intel Core 2 Duo CPU 2.20 GHz, with 2 GB of RAM.

6.2.2 Method Protocol & Research Questions

The aim of the thesis is to find a security testing approach capable of detecting malicious behaviours during the early stages of development and continuously assesses the security posture of the system's architecture while undergoing refinements. We have based our case study approach on the method defined by Yin Robert [199]. The object of study is the IS approach, which will be examined to measure its ability to detect malicious behaviours in the architecture and assess refinements to avoid breaking the existing security posture. We used single-case studies to observe representative systems for: (1) a web application, to ensure the approach's fitness for testing multi-threaded web fronts; (2) a cloud architecture, to test the approach with respect to uncertainties in the cloud; and (3) a distributed smart camera network with high dynamism. These case studies were chosen to allow us to observe *the effectiveness of the introduction of IS at the architecture level as a unit of analysis for testing the security of architectures in dynamic and unpredictable*

environments, in which emerging scenarios are probable due to unpredictable modes of execution. These types of applications are known to be challenging to test, compared to centralised applications, due to 'the problems inherent in deploying, controlling and monitoring many nodes simultaneously' [81]. The selection of applications varied, since we wanted to learn where the approach might perform better regarding threat detection or guidance of the refinement. These case studies were conducted by a group of two members to maintain the consistency of applications across different cases and ensure the correct application of method. One is a tester with security background, and one is an architect with testing background. The variables of interest are the total number of ISs, the number of ISs after refinements, which are program-generated, and the number of Security ISs, which are identified through a classification process (explained below) performed by the testers from the total set of ISs. The duration of the case studies averaged 4–6 months.

The questions we pose are:

1. Will the introduction of the IS approach aid the detection of security threats?

2. Will the introduction of the IS approach guide the refinement process to increase confidence in the architecture's resilience to threats?

3. Will the SecArch enhancement be superior to the IS approach in regards to exploring the search-space to find more threats?

4. Will the introduction of the IS approach guide the selection of architecture alternatives and candidates with respect to security?

The propositions we make are as follows:

- *Introducing the IS approach will allow for the detection of threats in the architecture.* This proposition will be measured by the number of Security ISs detected. If the case studies indicate that all the detected ISs do not affect security, then this proposition will not be satisfied.

- *Introducing the IS approach will guide the refinement to produce a more secure architecture.* This will be measured by the number of Security ISs detected before and after the refinement. The range of possible outcomes includes: (1) If no threats are detected after the refinement, then the refinement secured the design; (2) If the number of threats detected after the refinement is less than the number detected initially, then the IS approach produced a more secure architecture; and (3) If the number of threats remains the same or increases, then the IS refinements failed to address the security problems.

- *Introducing SecArch as an enhancement version to the IS approach will improve the search-space of threats, and it will result in the number of threats detected using SecArch being greater than the number of those detected using the IS approach.* This will be measured by the total number of threats detected in the architecture using both approaches.

- *Introducing the IS approach will guide the selection of architecture alternatives and candidates with respect to security.* This will be measured by the number of Security ISs detected in all architectures undergoing tests. If the architecture is found to have a lower number of Security ISs, then that architecture is considered to be less vulnerable to threats, and it has less potential for allowing unforeseen threats due to concurrent problems.

For the source of the scenarios, as a prerequisite of the approach to be used, either a list of specifications is available to model the required behaviour or the partial/prototype-/complete implementation of the application undergoing tests. In our case studies, the availabilities of the specifications and implementation varied. This allowed us to study different settings of the prerequisite information and determine a pragmatic use for the approach and its supporting tools. In situations where the specifications were available, we listed the scenarios required to fulfil the listed behaviours. In situations in which the implementation was present, we derived the scenarios from the code using the Visual-

Paradigm [135] tool to reduce the possibility of errors during the modelling process. This tool has won five major awards and is widely used in the industry (eg, Adobe, NASA) for UML modelling. All the derived scenarios were modelled using MSC specifications [84].

Once the system is modelled, the LTSA-MSC tool is used to search for ISs. When ISs are identified, they are collected and stored. They are then subjected to a classification process to identify which ISs have security implications. The classification process uses a taxonomy [70] that defines potential attackers, the impact of the threat, the objective of the threat and which resource is being targeted. Given the IS, the tester's role in the classification includes answering the following questions: (1) From the taxonomy's defined list of potential impacts (that is: increased access to resources, disclosure of information, corruption of information, denial of service and theft of resources), which of these may be achieved if the IS is executed? (2) Which of the listed attackers may have an interest in executing the IS? (3) Which resources would the attackers target? (4) What would be the attackers' objective? This information is stored in a septuple database along with the ISs' IDs, which highlight where the threat is found with graphical representation in an MSC form. When an IS does not have security implications, it is classified as a positive scenario, with no additional information. Testers use these positive scenarios to confirm the correct system behaviours.

The refinement process follows a systematic approach in which each threat is addressed in an order determined by the testers (eg, based on the severity of threats, due to timing constraints). When each threat is reviewed, a tester's role is to devise a refinement that prevents the occurrence of the threat. The prevention is confirmed when the IS does not appear in the following testing cycle. Each refinement is followed immediately by a testing process, allowing the testers to identify which refinements successfully prevented the IS and which caused the appearance of new ISs. A refinement may be retracted if it causes new ISs or, if necessary, it may be committed. When a refinement results in new ISs, these ISs are classified and further refinements are performed until either no additional ISs are identified or all ISs are positive. The data we collect in this stage are:

(1) the associated refinement (addition/removal/revision of functionality and changes to the hMSC graph); (2) the number of times refinements were made; and (3) the number of scenarios that arise (0 or more) for each refinement and how they are classified.

6.2.3 Summary of Applications

In the conducted case studies, we focused on using varieties of cases to test the effective use of the IS approach for security testing, such as the different types of applications, and different development processes. Table 6.1 discusses the case studies and their purpose in more detail, including the publication venues for each one.

Table 6.1: List of case studies carried out in this thesis.

Case Study	Type	Purpose	Impact & Research Questions
1	Web application	Studies the feasibility of using IS approach for security testing, and its application on web applications. The question we have answered in this case study is: how do we determine insecure behaviour in the system, while still maintaining observability of the problems associated with the behaviour, and also target implicit assumptions related to the design of the system? How well can IS approach guide the architecture refinement?	Published in ICSE-SESS'10. We presented to the software security community a novel IS approach for detecting compositional threats at the architecture-centric. We provided evidence in the study results. Answers questions 1 & 2.
2	Cloud	This case study presented the contributions of our work in extending the IS approach (SecArch) to detect hidden race conditions, allowing for evaluation of security with the presence of negative behaviour, and improving the system's security posture by refining the architecture to guard against potential hidden vulnerabilities. We demonstrated how an architecture-centric testing can assist in developing secure agile systems. We answered: Can we combine the IS approach with the agile development, such that both support each other to reduce conflicts? How can IS approach be aligned with agile practices to architect secure software systems?	Published in WICSA'12. We presented SecArch to the architecture community, and demonstrated its ability to provide a more holistic view of the provide a holistic view of the architecture. We provided evidence in the approach's ability to detect security emergent behaviour that is used to guide the refinement process to develop secure systems. Answers questions 1, 2 & 3.
3	Distributed System	The case study addresses the application of the IS approach on a distributed system. The aim of the case study is to test the fitness of the IS approach at guiding the selection of secure architectures. We have also studied the approach's architecture refinement support with presence of trade-offs between security and utility.	Provides a novel application of the IS approach for distributed systems, where we test the fitness of the IS approach to guide the selection of secure architectures. Answers questions 1, 2 & 4.
4	Identity Management (Appendix B)	This case study is conducted twice by two masters students. The aim of the study is to extend the confidence in the IS approach, as well as examine how the IS approach maybe used in practice, in terms of flexibility and ease of use. The question of the study was to examine which of the two identity management architectures should be chosen in terms of security.	Book Chapter by Elsevier. We presented the iterative and incremental features of the IS approach to the test-driven community, and demonstrated its ability to guide the the development of agile systems. Answers questions 1, 2 & 4.

6.3 Case Studies Applications

In this Section we will cover three case studies. These studies show new modalities of applications. We have also included Case Study 4 in Appendix B to further illustrate applicability, potential replication of the steps, etc. The case study method is described in Section 6.2.2.

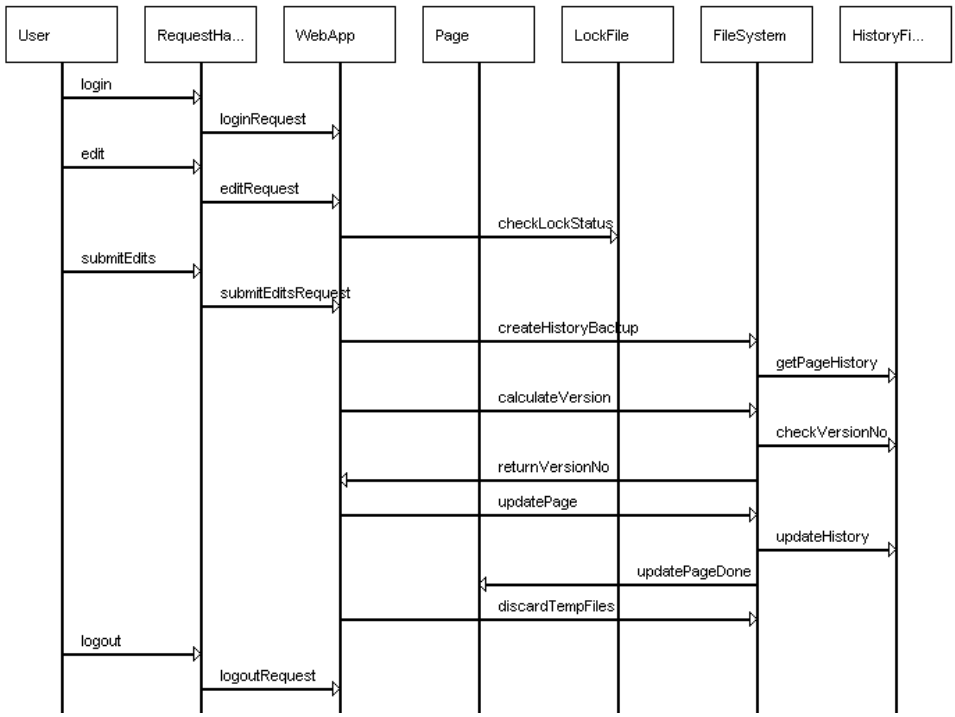
6.3.1 Case Study 1: Web application

In this case study, we proposition that the introduction of the IS approach will (1) allow for the detection of threats in the architecture, and (2) guide the refinement to produce a more secure architecture. The case study application [33] attempts to simplify the process of editing HTML pages as done in Wiki or blog software (eg, uploading a figure, editing a page, log in and out, etc). We have chosen this case study for the purpose of: (1) testing the approach's applicability to concurrent web applications where there is a shared resource; and (2) studying the approach when the scenarios are not present upfront (ie, only the implementation). This allows us to realise the limitations of the approach when applied to a realistic environment. The number of testers who carried out the study is two. The method protocol we used in this case study is described in section 6.2.2.

6.3.1.1 Scenario Modelling & IS Detection

In Chapter 4 we explained the approach in details. In order to detect ISs, we need the set of specified scenarios representing desirable functional behaviours, and the graph (hMSC) representing the continuation of the scenarios (ie, how these scenarios combine together to fulfil the overall system behaviour). Because we only had the implementation and the list of functionalities provided by the application, we extracted the scenario behaviours from the code. The set of scenario names and descriptions are presented in Table 6.2. The scenarios presented allow a user to login, edit a webpage if the page is unlocked and request a lock on the page if he is an administrator. The desired behaviour ensures that access

Figure 6.1 An acceptable scenario of various possible scenarios in the web application. The web application receives a request to edit a webpage, where it checks if the user is authorised. Once the user commits any changes to the webpage, the application creates a history backup, a version number, and updates the webpage.



control is maintained on all web pages. We modelled the scenarios in MSC specification (eg, Figure 6.1), and their directed graph (hMSC) that shows possible continuations of the scenarios to illustrate the desired global behaviour of the system. The scenarios and the hMSC models were fed into the LTSA-MSC tool to search for and detect ISs. Figure 6.2 shows the hMSC model, which shows the continuation of scenarios that have two main behaviour branches:

1. Users to edit the displayed web page after checking that the web page is not locked, and

2. Administrators to set a lock on the web page to prevent users from editing the page.

When we composed the scenarios together, we detected sixteen emergent behaviours (ie, ISs) in the architecture. Each of these scenarios was given an ID and collected in a database file as a graphical MSC.

Figure 6.2 The hMSC graph for the scenarios in Case Study 1 web application. It shows how scenarios are integrated together to make up the overall desired behaviour.

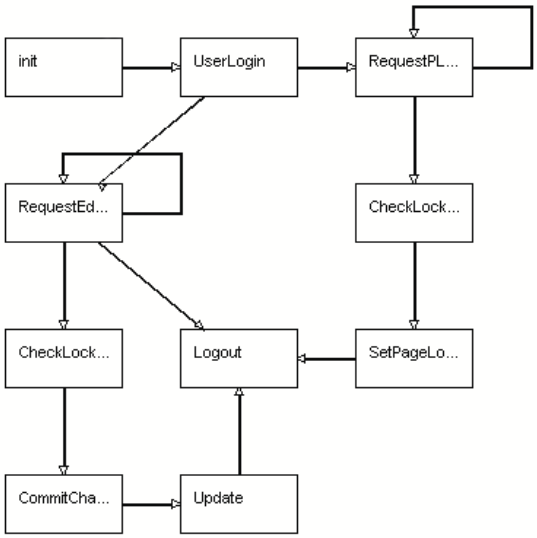


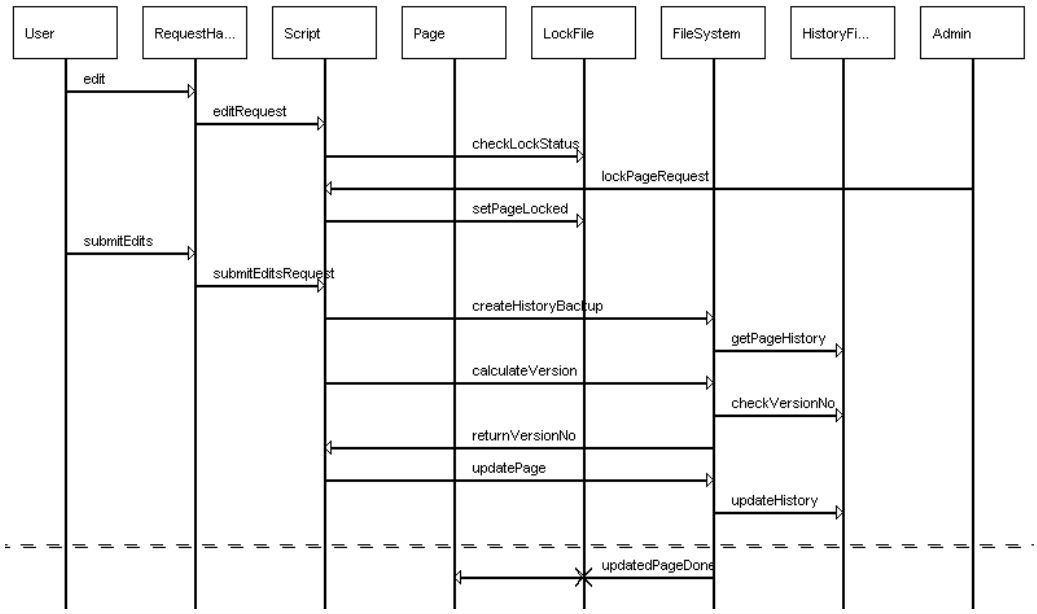
Table 6.2: Scenarios elicited for Case Study 1: Web application.

	Scenario Name	Scenario description
1	UserLogin	Allows users to login
2	RequestEdit	Records edit request
3	Update	Commits changes to pages
4	LockPage	Sets pages locked
5	RequestLock	Records requests for editing
6	Logout	Allows users to logout
7	LockCheck	Checks the lock status of the page
8	Authenticate	Checks users authority to edit pages
9	CalculateVersion	Determines the version number of the next edited page

6.3.1.2 Classifying ISs

The classification process was carried out by two testers as explained in Section 6.2.2. Given the detected sixteen ISs, we found six of which were Security ISs. The classification results are depicted in Table 6.3. Given each IS trace in the form of an MSC scenario (eg is shown in Figure 6.3), we are presented with the precise behaviour of the system that was not anticipated before the scenarios were composed. The classification task was to understand how this behaviour can be abused to launch an attack on the application. We use the classification scheme [78] presented in Chapter 4 to explore various ways in which that trace can be misused. The classification scheme [78] states that there are

Figure 6.3 IS detected in Case Study 1 web application. Happens when the Admin requests that the page is locked after the system had checked that the user is authorised to edit the page, which subsequently allows the user to continue to edit the page.



five unauthorised security results of attacks: *increased access to resources, disclosure of information, corruption of information, denial of service and theft of resources.* We used this information to consider the ways in which the execution of that trace could result in loss of data or cause denial of service, and what resources may be targeted. The classification scheme also lists the types of attackers that may execute the trace. Reflecting on the taxonomy, the types of attackers we found who may attempt to break into the web application are:

- hackers - attack computers for the challenge, the status or the thrill of obtaining access.
- vandals - attack computers to cause damage.
- voyeur - attack computers for the thrill of obtaining sensitive information.
- spies - attack computers for information to be used for political gain.
- corporate raiders - employees (attackers) who attack competitor's computers for financial gain.

Table 6.3: Detected Potential Attacks found in the Case study 1 web application

	Attack Name	Attack Category	Target resource	Hacker type	Objective
1	Unauthorised edit after a page lock	Increased Access to resources & Corruption of Information	Pages	voyeur, hacker	adds/removes sensitive data
				vandals, corporate raiders	adds false/misleading information
2	Overwriting previous edit	Corruption of Information	Pages	hacker, vandals, corporate raiders	hides data for personal/ political gain
3	Sensitive data loss due to malicious page locks	Corruption of Information & Disclosure of Information	Pages	hacker, vandals, corporate raiders	Causing intentional loss of data
4	Unauthorised edit after a role is revoked	Increased Access to resources & Corruption of Information	Pages	voyeur, hacker	adds/removes sensitive data
				vandals, corporate raiders	adds false/misleading information
5	Re-saving deleted page	Disclosure of Information	Pages	hacker, vandals, corporate raiders	either reloading hidden data to reveal sensitive data, or cause inconsistency
6	Overwriting history files	Corruption of Information	History files	hacker, vandals, corporate raiders	hides data for personal/ political gain

The results of threat detection by this classification are presented in Table 6.3. A general observation of the taxonomy for categorisation of attackers is that the names reflect on the objective of the attack. It opens up options as to what the application might be used for (for example, the pages might be used for political purposes); thus, allowing us to think of a broader range of misuses of the application, which may be unknown in advance (who will use the general-purpose applications?). In the above case, the pages can be used for many different purposes: from personal purposes to writing version information about software to political usage. We can also see that the pages may be edited by different roles with different access rights. The realisation of different objectives behind the attack, and the roles involved, gave clues to abusive behaviour.

6.3.1.3 Security ISs Results

In this section, we will breakdown the results of the detected ISs in details, their destructive execution outcome, who may initiate the attack, and how it is accomplished.

- IS 1 (unauthorised edit) occurs because although the administrator has requested a lock on the page, the user still successfully edits the page. This violation occurs because as far as each individual process can locally tell, the scenarios are proceeding according to the given specifications (namely, scenarios 2-3 and 4-5). That is: (1) the user is checked for authority and it is ensured that the page was not locked when

the user edited the page; and (2) the administrator locked the page. As these two scenarios interleave, the result is not as expected when each is executed on its own. This IS (shown in Figure 6.3) models a vulnerability in the system that permits: (1) an access control violation since the user gets access to modify the web page even after the administrator has requested a lock on it due to timing conflicts; and (2) data integrity violation since the state of the data would have been modified after the administrator had requested the lock on the page.

- IS 2 (overwriting previous edit) occurs when two users attempt to perform an edit operation at the same time. The application attempts to handle concurrency by: (1) creating a backup file of the web page; then (2) writes the results back to the history file with the appropriate version number; and (3) displays the last version of the page. Such an operation fails to perform as expected because the application generates the version number from the last version number created in the file. The IS detected presents a possibility of the same version number being issued in both processes if each requests the version instantly. Thus, when two users request a version number, the application checks the last backup file to determine the next incremental version number, and mistakenly sends it to both users. Eventually, one of the users overwrites the results of the other because his submission contains the same page number and version. The security consequence in the system is a loss of data, which might be intentional, seeking to cause a denial of service attack in the system or to hide data that holds evidence against one of the users.

- IS 3 (sensitive data loss due to malicious page locks) occurs when a user submits sensitive data, and because a page had been locked by the admin, data transported over the network is lost and an error is returned. An administrator may lock pages on a temporary basis; for example, during maintenance of the pages, a malicious administrator may intentionally lock pages to cause loss to the company. This privilege overpowers the admin and it violates the security's 'principle of least privilege'

[154], where every process or user should operate using the least amount of privileges necessary to complete the task. In addition, errors are a common source of disclosure of sensitive information, where the system fails to terminate correctly, leaving data unhandled in the memory.

- IS 4 (unauthorised edit after a role is revoked) is similar to IS 1; both IS threats model the behaviour of a user attempting to modify a web page. In IS 1, the system administrator changes the state of the web page to make it locked (hence, uneditable), and in IS 4 the administrator changes the access rights assigned to the user. IS 4 occurs because the application does not mediate access to the resources during checks and during usage. This breaks the security principle of 'complete mediation' [154], in which every access to an object must be checked for authority just before its use. The security consequence in the system is that users gains more privilege than is authorised, which results in the violation of data integrity and loss of data.

- IS 5 occurs when a user edits a page that is concurrently being deleted by the administrator, which is then saved by the user. Because the user holds a temporary copy of the original file, the temporary file is then written back into the system, though the administrator deleted the original copy. This behaviour could allow disclosure of information if the deleted page contains secret information.

- IS 6 (overwriting history files) is a variation of IS 2 but with more impact. When the user can mistakenly overwrite a previous edit due to an incorrect version number system, then a malicious user can submit maliciously formed pages (with a forged name and version number) to overwrite previous history files. This is not a straightforward attack; for it to be successful, the attacker would need to know the version numbers of all pages he intends to corrupt. Note that the pages will not be deleted, because the user does not have the authority to delete.

Table 6.4: Refinement summary for Case Study 1

	Refinement	IS detected
IS 1	Added function: checkLockStatusAndLock and modified the architecture flow.	3 positive IS
IS 2 and 6	Removed function: calculateVersion	0
IS 3	Added internal method to store user data in data Sessions or alternatives	N/A
IS 4	Added function: authenticate	1 positive IS
IS 5	First refinement: Added function: checkOriginalPage	1 Security IS
	Second refinement: modified architecture flow to temporarily lock the original file to be edited.	0

These attacks are similar in the sense that they occur because each process is unaware of the global behaviour of other components, and instead depends on local information. The approach was able to detect four types of threats out of five presented in the classification scheme [78].

6.3.1.4 Architecture Refinement Results

To perform a systematic refinement process over the detected threats, we will revise each threat one at a time; and as we modify the architecture model, we will retest the architecture to ensure that the refinement did not introduce new, unknown behaviours. We store every detail about the refinement (eg, revisions, additions) This step will prevent the breaking of the security of the system as new refinements are performed. It is possible to refine several related vulnerabilities in one step if the refinement will address all vulnerabilities using one fix (as we did with ISs 2 and 6 in this case study) to speed up the process; however, it is not recommended to apply all changes at once, nor unrelated vulnerabilities, because this will make it harder for testers to identify the source of the problem. The refinement summary is presented in Table 6.4.

1. (Addressing IS 1): In order to prevent an unauthorised edit after a page lock is executed, the locking mechanism needs to be refined to address its fundamental design limitation. Rather than performing the authorisation of an edit in one place (during the edit request), we can instead check the authority just before the edit

is committed so that, if an administrator locks the page after an edit is requested, the system can double-check the authority at the second time for final confirmation, and write-lock it until we finish writing the file. To refine the architecture, we needed to insert the second check into the model, and add a modified version of the checkLockStatus to perform a temporary write-lock when an update is committed. We then retested the architecture for new ISs. The IS approach detected three new positive ISs that do not impact the system. Thus, we committed the refinement of this phase.

2. (Addressing ISs 2 and 6): Rather than determining the next version number with respect to the last edited file, a version can be provided on the fly based on the date and time the server machine recorded while committing the edit requests. This can be calculated as the last step in committing new changes to the pages. This synchronised operation will guard against multiple-versioning and will prevent the overwriting of files because each version will depend on a specific moment in time of the server machine. The refinement of this IS involved removing the calculateVersion method from the interleaving scenarios, instead making an internal call in the application to calculate the version based on its date and time. Unfortunately, the internal call cannot be modelled using the IS approach, because ISs are not concerned with internal behaviour; they are instead concerned with the observable communication between components (ie, message sharing). Thus, the limitations do not effect the IS's detection ability. The refinement result is successful; it prevented the ISs detected in the original architecture model, and no new IS appeared as a result.

3. (Addressing IS 3): To prevent sensitive data loss, if the user editing a page is authorised, we need to include a functionality allowing for temporary storage of user data before the data is transmitted over the network. In situations where an error takes place, the user will be able to restore his data and attempt another

submission. This can be done by storing the user data in temporary data structures (such as sessions) that can be retrieved in situations where an error takes place. This refinement cannot be modelled because it is an internal call within the web application.

4. (Addressing IS 4): To prevent a user from editing a file after being revoked, we have applied the same principle as the first refinement except that the check takes place for the user as well. This means that the user is checked for authority twice before he can make a change to the page. To refine the architecture, we needed to insert the second check on the user-that is, to 'authenticate' him- to perform a user status check before the file is temporarily locked when an update is committed. We then retested the architecture for new ISs. The IS approach detected one positive IS that did not impact the system. Thus, we achieved the refinement of this phase.

5. (Addressing IS 5): To guard against the user rewriting a deleted page, we need to commit new versions of pages only in the condition that the original page exists. This means that we need to add another functionality that checks the existence of the original file being edited. However, this refinement caused another IS where it was possible between the check operation and the rewriting of the file, and the original file was deleted. The second refinement involved performing a temporary write-lock on the original file until it is rewritten, and then releasing the lock. This will guard against two problems: (1) rewriting deleted pages; and (2) a race condition between the check and the writing of the new version. Retesting for ISs did not detect emergent behaviour.

To summarise, from the original architecture model and the six ISs detected, through refinement, we were able to move to a more secure architecture that only exhibited positive behaviour. Figure 6.4 represents the new hMSC model of the architecture. As the refinement continued, we were able to confirm that the changes were acceptable before we committed to the refinement change. We found that the original refinement to address IS

Figure 6.4 Moving from less secure architecture, to a more secure refined architecture for case study1 after the changes in Table 6.4 are applied.

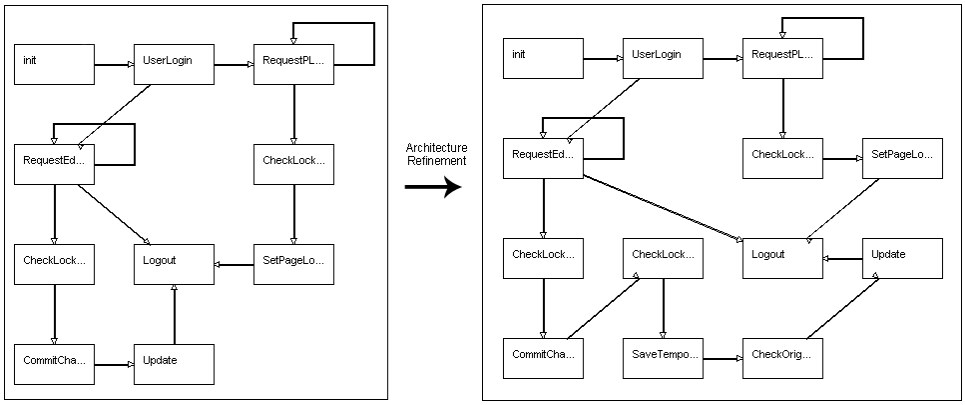
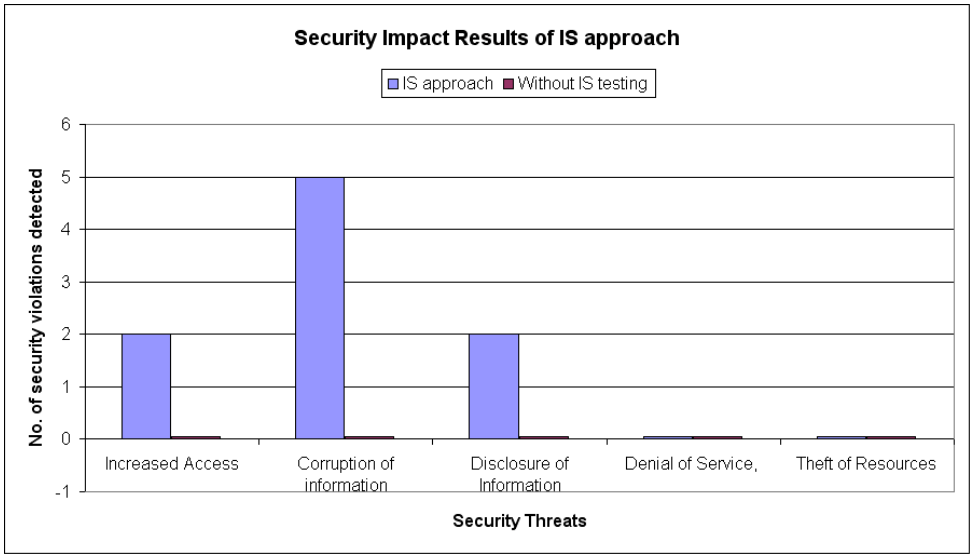


Figure 6.5 Security Impact of the IS approach for Case Study 1. The x-axis values are gathered from the classification results presented in Table 6.3.



5 was not secure, and we considered another change before we accepted the final architecture. We have also seen that this process is systematic and guided by the approach. It is possible to first prioritise the threats based on the security impact of each one using the classification scheme presented in Section 6.3.1.2 to overcome any time (budgets do not help). More importantly, the refinement was done incrementally, and the approach responded to refinement changes by searching for additional behaviours that may have appeared as a result of the change.

6.3.1.5 Summarising the Findings of Case Study 1

In this study, we propositioned that the introduction of the IS approach will (1) allow for the detection of threats in the architecture, and (2) guide the refinement to produce a more secure architecture. We have mentioned in section 6.2.2 that the satisfaction of the first proposition is measured by the number of Security ISs detected.

- In this case study, we detected sixteen ISs, of which six pose threat behaviours that the attackers may perform to violate the security of the application. The graph presented in Figure 6.5 shows the security impact with and without the IS approach. We can read from the graph that the approach detected more sources for information corruption due to the nature of the application domain. The other 10 positive ISs were used to check the architecture's conformance with the desired functional requirements. These positive traces provide an extended view of the functional requirements because we are no longer limited to viewing the functional requirements as single, individual scenarios; instead, we are now viewing the runtime behaviour of functional requirements to verify that the functionality is used correctly. Our inspection of the 10 traces confirmed the correct usage of scenarios. The six Security IS traces were reviewed in details Section 6.3.1.3.

- We have also mentioned in section 6.2.2 that the satisfaction of the second proposition is measured by the number of Security ISs detected before and after the refinement. We have seen in Table 6.4 that the refinements were either solving the security threat for which no new Security IS arise (as in the refinements of ISs 1, 2, 3, 4, and 6), or that less than the initial six threats we started the refinement (as in IS 5).

The conclusion we make is that, the propositions we have set in the initial case study have both been satisfied, and that from these preliminary results provided us with very useful insight into the security posture of the application and, in particular, that the IS approach seemed to be well-suited for the detection of potentially corrupt information.

6.3.1.6 Discussion

We can observe that the use of the IS approach detection in security testing is beneficial to complement the limitations in the existing security testing approaches (as discussed in Chapter 2). The reason as to why existing techniques are unlikely to detect these threats is that existing approaches execute scenarios in a sequential order. In contrast, the IS approach takes into account the concurrency nature of multi-threaded applications, and studies the occurrence of unexpected interactions between components. This case study demonstrates the ability of the IS approach to detect emergent behaviour that affects the security posture of the application. Incorporating the IS approach in the security testing process will help testers to automate the process of detecting threats on the architecture, which might be overlooked manually [185, 5]. We can also observe the advantage of the IS approach that allows testers to think outside of the specifications. Such knowledge is very important for security, since attackers target implicit assumptions made by designers. Because the achievement of software security will only happen if security testers detect threats before attackers exploit them, the results in this case study indicate that this approach is proactive in detecting malicious behaviours in a given architecture, and will allow us to protect the software before attackers exploit its weaknesses. The case study validates the threat-detection capabilities of the IS approach and its application for security testing.

Another angle of interest deals with the concept of scenarios used in the IS approach that allow us to move back and forth between the functional requirements and the architecture. The architecture modelled the composition of the requirements presented in Figure 6.2, and its traces represented the message ordering of the scenarios in a composite manner. This means that each trace models one execution of the architecture from start to finish. A trace example is: $login > loginRequest > lockPage > forwardLockRequest > checkLockStatus > setPageLocked$. These sequences of traces allowed us to verify the conformance of the architecture with the initial set of scenarios (with respect to the overall desired behaviour when the requirements were composed). The IS approach goes through

all possible traces in the architecture (the desired traces, and the detected ISs).

The advantage of an IS is that it allows us to replicate dynamic behaviours to detect concurrent faults that happen at very specific moments in time. Attempting to find concurrent faults in live systems can be technically challenging, because there is no guarantee as to how long the application must be run before an emergent behaviour is detected. For example, when we test the usage of 100 users at the same time, the testing process may not reveal the condition where two users commit a simultaneous action. These conditions are hard to identify using random live executions unless they are specifically tested for. On the other hand, IS allowed us to systematically identify the condition to prevent the possibility of overlooking critical conditions.

6.3.2 Case Study 2: Cloud application

In this case study, we make three propositions about the introduction of the IS approach: (1) it allow for the detection of threats in the architecture, (2) it will adapt to changes in agile-like developments, and (3) introducing SecArch as an enhancement version to the IS approach will improve the search-space of threats, and it will result in the number of threats detected using SecArch being greater than the number of those detected using the IS approach. The case study aimed at testing the potential of the approach for testing the security of architecture interfacing with the cloud, a dynamic and unpredictable environment where emerging scenarios are probable due to an unpredictable mode of execution. This case study was conducted in three stages. The first stage was aimed at testing the potential of the approach and its applicability to an industrial setting. The application also aimed at soliciting an unbiased view of the approach's potential when applied by a industrial-party architecture team (comprised of two senior architects, an MSC student and 2 developers). The results of this stage were published in [53]. We built on the results of the first stage to evaluate the adaptivity of an IS approach in which we added more functionality. We observed that the IS approach adopted the new functionalities and revealed new ISs. In the third stage, we applied SecArch (an extended

approach to the IS as described in Chapter 5) to the outcome of Stage 2 in order to test its effectiveness in detecting additional ISs. The application revealed further ISs as documented in Section 6.3.2.4. For all these stages, we followed our method protocol described in Section 6.2.2.

In the following Sections, we will: (1) introduce the case study; (2) briefly discuss the method and results of the industrial case study as a background to our case study; and (3) compare the outcome of both case studies with respect to the IS approach. We will then show (4) the application of SecArch to the rest of our case study.

6.3.2.1 Industrial Cloud Case Study Background

The case study was provided by Xactium Limited[1]. The purpose of the case study was to develop a secure architecture that interfaced between a bank and the cloud environment as a means of provisioning risk management services related to the bank's business [53]. The bank had a proposal for adopting the prominent SaaS cloud provider, Force.com, to process their risk data. The design of the architecture had undergone many changes, and due to privacy concerns, this case study was performed on the initial design of the architecture only. The case study consists of 36 files with total number of 2739 lines of code. The desired functionality as identified by the industrial-party form 11 functional scenarios that need to be handled securely by the architecture as shown in Table 6.7: (1) synchronising data across local databases and the cloud; (2) registering new users to the secure architecture; (3) subscribing users to cloud applications; (4-5) refining private fields and decrypting/encrypting of new fields; (6) user withdrawals; (7-8) viewing secure and non-secure data (including decryption); (9) revoking compromised keys; and finally (10-11) submitting secure and non-secure data (including encryption). The architecture consists of seven components with two types of users: registered users and administrators.

[1] http://www.xactium.com

6.3.2.2 Stage 1: Industrial-Party case study

The case study was conducted on an industrial case study with the involvement of architects from Xactium and Salesforce.com. The case comprises an activity to evaluate the architecture of a bank application for security when deployed in the cloud. The industrial-party where provided with the necessary details to use the tool correctly, where we held 30 minutes session explaining the tool. We also introduced them to the method protocol, the data collection plan, and how to evaluate the outcome of the results, which we included in Section 6.2.2. The case study began after the tool was thought to be simple to use by the group. The scenarios of this case study and the desired hMSC were designed by the industrial-party. We have then held monitoring sessions to observe their usage to ensure it remains correct, and that the data is collected consistently. The outcome of their case study demonstrated the new modalities of IS applicability to industrial setting. They presented an architecture evaluation approach suitable for the dynamic unpredictable environments [53], such as the cloud. They have combined the benefits of ATAM and IS for evaluating the security quality attributes of architectures in this domain, where they have addressed weaknesses in the static analysis architecture evaluation method (ATAM) by enriching it with innovative ideas from a dynamic analysis method (IS approach) to generate subtle scenarios which may lead to security attacks on the architecture if undetected. Their results indicated that their combined approach found additional security scenarios beyond the plain ATAM, resulting in new risks and trade-off points. In particular, they concluded that the IS approach was able to detect critical security scenarios that were not captured with the use of static analysis ATAM alone.

Settings and Results of Industrial-Party case study Their study covered scenarios: (4-5) viewing secure and non-secure data (including decryption); and (7-8) submitting secure and non-secure data (including encryption) to the cloud. The first ISs detected corresponded to the case in which the client makes a web service call to encrypt a sensitive field value but an attacker is able to manipulate the web service to perform a decryption

Figure 6.6 IS attack reported by Funmilade et al [53].

- Bank staff member sets out to encrypt a sensitive data field (eg, Net Loss value).
- Web service queries Security Manager for sensitive status of this data field.
- Web service passes the request to the Security Manager.
- Security Manager returns result indicating that the data field is sensitive.
- Instead of calling the encryption component, the malicious web service consumer calls the decryption component and passes the pre-recorded ciphertext to it as parameter value.
- The web service returns the actual value of the Net Loss in plaintext.

Table 6.5: Industrial-Party case study results. Results indicate a detection of 2 ISs of which one affects the security status.

Scenario No.				IS detected	Positive	Security IS
Scen4	Scen7	Scen5	Scen8	2	1	1

operation instead. Such a vulnerability could be exploited to perform a key replay attack on sensitive fields. This attack could be achieved by first requesting an encryption of a sensitive field value, and then subsequently requesting a decryption with some previously compromised key. The detected IS is not trivial, as this represents a security risk that could lead to further security scenarios when examined from a technical/business perspective; hence, it was classified as high-risk. Their description of the attack is presented in Figure 6.6. The IS detected in the first phase of the evaluation was added to the top scenario list of the ATAM and the architecture was subsequently revised to address it. Table 6.5 presents the results of the industrial findings.

Discussion The observation we gain from this case study is that the IS approach was used to complement a manual evaluation process (ATAM), which did not detect the Security IS threat reported by our approach. This shows how the IS approach can provide better insights when evaluating dynamic and unpredictable environments, because it focuses on the dynamic behaviour that may not be visible at the static level of the communication (that is used in ATAM).

6.3.2.3 Stage 2: Testing Adaptivity

Software is made to evolve in response to changes in its context and requirements. Changes may be needed to reflect business needs, new regulations, faults or corrections. As the system evolves, security concerns need to be analysed in order to evaluate the impact of changes in the system [182]. Changes in functionality may introduce security-related vulnerabilities that need to be re-analysed to investigate the impact of the change in the system. The issue with maintaining security while introducing change is thought to be challenging due to the ad-hoc nature in which changes are handled [182].

In this part of the case study, we proposition that the IS approach will respond to changes in the functionality of the architecture to allow testing agile-like architectures. We test the satisfaction of this proposition by studying the ability of the approach to (1) testing the addition/removal of new sets of functionalities to an existing iteration (possibly created by another team), and (2) combine two completed iterations together. We consider adaptivity with respect to agile software development [95] because agile development is an iterative, incremental approach to developing and releasing software. Agile principles include commitment to timely and ongoing software deliveries, changing requirements, simplicity in approach and sustainable development iterations. Its practices include frequent releases, ongoing testing and stakeholder participation throughout the development process. We thus proposition that the application of the IS approach at the architecture level supports the principles of agile development [96]. We aim to show that the timeliness of the approach supports modern/current development processes.

In Section 6.3.2.1, we looked at the set of desired functionalities in the form of scenarios of the case study, and we have seen that the industrial-party case study has tested the composition of four of the core functionalities. We consider this case study to be the first iteration of agile development, and choose to continue with the second iteration, which involves adding the seven additional functionalities to the existing set. The case objective is to exemplify how IS can support agility when the teams developing and testing the iterations are different.

Composing Scenarios Due to limitations in the scalability of the LTSA-MSC tool, composing all scenarios together caused state explosion. To solve this, we have separated the scenarios (shown in Table 6.7) into groups as shown in Table 6.6. Grouping of scenarios took into account that:

- Determined attackers rely on probing an unexpected sequence of actions [177]; thus, inspecting different compositions of scenarios gives better coverage to account for possibly unpredictable modes of use; and that
- Testers' assumptions about the system usage may be subjective and may not represent the real usage.

To balance the above two considerations, we have chosen certain compositions of scenarios that convey meaningful paths of possible interactions that may potentially cause problematic compositions, such as the simultaneous accessing of a shared resource like a user key or a local database. For example, Scenario 2's 'User registration' and 3's 'Subscribe' are sequential processes that occur at the start of the user's processes, and prevent the user from performing any other operations until both are completed. Furthermore, Scenario 9's 'key revoke' is only relevant when we are dealing with encryption-related scenarios (such as Scenario 4). We also wanted to test rare compositions of scenarios that may be triggered by attackers, such as an administrator interfering in the user's behaviours. Our two-dimensional approach supports a balanced test coverage with minimised subjectivity that is tailored to suit security-testing requirements.

Initial Results of Adaptivity Once the scenarios were readily composed, the search for ISs yielded a total number of 22 ISs, of which 7 of these were classified as Security ISs as shown in Table 6.6. The table reflects a successful integration between the first iteration conducted by the industrial-party case study (shown in Table 6.6 as Combo6), and the rest of scenario combinations 1-7. These results indicate two aspects of testing:

- We were able to test the system without direct interaction with the first agile iteration (Combo6), as shown in the scenarios of Combo1 in Table 6.6. This would allow

the testing of a coherent set of functionalities that may not interact with other sets of scenarios.

- We have also tested different levels of interactions with the first iteration (as in combinations 3 and 4) in order to study the results of adding new functionalities into the system, and to investigate how the additions can break the security of the first tested iteration.

Thus, the approach reflected flexibility in testing either the development of a new iteration or the integration of existing iterations together. Therefore, the IS approach satisfies the proposition that the IS approach will respond to changes in the functionality of the architecture to allow testing agile-like architectures (as we stated in Section 6.3.2.3). This is of particular importance for the component-based development industry, where different companies might create components at different times; thus, testing components individually as well as their integration with other components is an essential part of testing. It also shows how the approach is test-driven and focuses on reusing and testing the integration of components built iteratively, and ensuring that when components are created, a correct overall behaviour can be maintained.

We can also read from Table 6.6 that the testing of new functionalities triggered new emergent behaviour that needs to be analysed before the system is developed further. The last column in Table 6.6 presents the number of Security ISs detected on different scenario combinations. We can see that scenario combinations 2 and 3 (with a total of 6 and 4 ISs), have more gaps in their specifications than combinations 1 and 6 (with a total of 1 and 2 ISs). This indicates that when scenarios are closely related but do not have strong synchronisation between processes, different behaviours might result. For example, if we consider scenarios in Combo3, where we have a user manipulating sensitive data and changing the security settings of fields, which motivates an administrator to revoke that user, then the results can vary; for example, either the user still manages to corrupt data, or important data might be lost due to the clash between the two actions.

Table 6.6: Application of the IS approach in the second development iteration of Case Study 2. The results show the detected ISs in the combination of scenarios. These results indicate that the seven additional functionalities were adapted into the existing set (as studied by the industrial-party shown in Section 6.3.2.2), and that we were able to reveal new ISs from different compositions of scenarios.

	Scenario No.				IS Approach	Security IS
Combo1	Scen10	Scen9	Scen1	Scen2	1	0
Combo2	Scen3	Scen6	Scen5	Scen1	6	1
Combo3	Scen5	Scen4	Scen6	Scen10	4	1
Combo4	Scen1	Scen11	Scen10	Scen6	2	0
Combo5	Scen3	Scen11	Scen6	Scen5	3	1
Combo6	Scen4	Scen7	Scen5	Scen8	2	1
Combo7	Scen9	Scen1	Scen10	Scen11	4	3
				Totals	22	7

Due to the size of the results, the details of the 7 Security ISs can be found in Appendix A. Given the nature of the application and its clients, we found that the types of potential attackers on the software are:

- corporate raiders - employees (attackers) who attack competitors' computers for financial gain.
- vandals - attack computers to cause damage.
- voyeur - attack computers for the thrill of obtaining sensitive information.
- professional criminals - attack computers for personal financial gain.

6.3.2.4 Stage 3: SecArch Case Study

In Chapter 4, we discussed the conceptual importance of the IS approach in detecting security vulnerabilities, and we discussed the negative effects of inconsistencies and hidden interactions on the security of software from a single behaviour-view. In Chapter 5, we presented an incremental architecture-centric approach for security testing (SecArch), which merges the IS approach and race condition analysis techniques to systematically guide testers in detecting vulnerabilities and evaluating the architecture's security posture.

This method extends the IS approach to enhance its search space such that we can discover more emergent behaviours. We have found [6] that even though we detected all possible ISs in the behaviour model (ie, reached the assumption of completeness), we were not able to detect the scenarios we aimed to discover.

For this part of the case study, we propositioned that the introduction of SecArch as an enhancement version to the IS approach will improve the search-space of threats, and it will result in the number of threats detected using SecArch being greater than the number of those detected using the IS approach. This proposition stems from the belief that incorporating structural and behavioural views of the architecture can enrich the search space to allow us to detect further malicious and emergent behaviours. Smaller search spaces may imply false senses of security in which a tester believes the system is free of emergent behaviour, when in fact the search space itself was too limited to find more behaviours. We have demonstrated in Chapter 5 Section5.1.1.4 that the IS detection algorithm builds on strong assumptions about the ordering of events, such as only supporting single-queues (ie, enforcing visual order). Restrictions on queues make for unrealistic assumptions regarding the components involved in the concurrent system. To address this limitation, we have proposed the integration of another algorithm (which we refer to as the Alur's algorithm [14])), that addresses the detection of race conditions with respect to the timing constraints of the message, the queuing structure (such as FIFO), any possible semantic interpretations and can detect conflicts such as causality cycles and race conditions. We use Alur's algorithm because it allows us to check for race conditions based on several queue structures, filling the gap in the IS approach (this is discussed in greater detail in Chapter 5).

Initialising the Case Study By initialising the case study before we started testing the integration of the IS approach with Alur's algorithm, we wanted to test if any of the scenarios had inherent design flaws. This allowed us to realise what was detectable by either the IS approach or Alur's algorithm, and what was detectable only by the SecArch

Table 6.7: System scenarios tested using IS approach [185], and Alur's algorithm [15]. Results indicate no ISs detected on each individual scenario, and 1 race condition found in scenario 9.

	Alur's algorithm	IS approach
Scen1: Synchronise	0	0
Scen2: User Registration	0	0
Scen3: Subscribe	0	0
Scen4: Set Fields Encrypted	0	0
Scen5: Set Fields Decrypted	0	0
Scen6: Revoke user	0	0
Scen7: View Regulated Data	0	0
Scen8: View Plain Data	0	0
Scen9: Revoke Key	1	0
Scen10: Save Regulated Data	0	0
Scen11: Save Plain Data	0	0

enhancement. Checking each scenario independently could reveal whether the component had a limited view and needed to be synchronised with other components in that scenario. Table 6.5 presents the results of testing all of the scenarios individually for IS and race conditions. Only the 'Revoke Key' scenario contained a potential race condition. In the coming Sections, we will see that providing an incremental approach when searching for race conditions ensures that hard-to-detect race conditions are dealt with before the system is built.

Table 6.8: Classifying Race conditions and ISs detected in Case Study 2.

	Threat	Attacker	Objective	Results	Target Resource
1	RC1	vandals, corporate raiders	Using wrong key to encrypt	Corruption of information	User Sensitive data
2	RC2	vandals, corporate raiders	Overwrite sensitive data	Corruption of information	User Sensitive data
3	IS1	vandals, corporate raiders	Corrupting sensitive data	Corruption of information	User Sensitive data
4	IS2	Vandals	Damage	Corruption of information (inconsistency)	Local database and cloud
5	IS3	voyeur, hacker, professional criminals	Gain access to sensitive data	Denial of Service, Disclosure of Information	Webservice

Demonstrating Results Table 6.9 shows that a total of 21 Security ISs were detected, 14 of which were only detectable by SecArch and the other seven we detected using

the IS approach as shown in Stage 2 of the case study in Section 6.3.2.3. To come to the conclusion that these ISs are security threats, we have classified the test results, including the potential damage and the objectives behind the attacks (for readability we provide a summary of Combo1 threats in Table 6.8, full details of which can be found in Appendix A). Even though the classification scheme allowed variations of attackers and objectives, which helped us to envision threats, we have experienced two limitations in its classification. We recall that the classification of unauthorised results was limited to 'Increased Access, Disclosure of information, Corruption of information, Denial of service, and Theft of resources'. There were situations in which a description of the results of the threat using these terms was not sufficient; for example, the situation that arose when we needed to describe an inconsistent situation that did not disclose or corrupt information. We also experienced two instances of damage that could occur without direct intervention from an attacker. We wanted to model an *insider* that was not necessarily from a corporate company. Thus, we have extended the meaning of 'professional criminal' to include both outsiders and insiders.

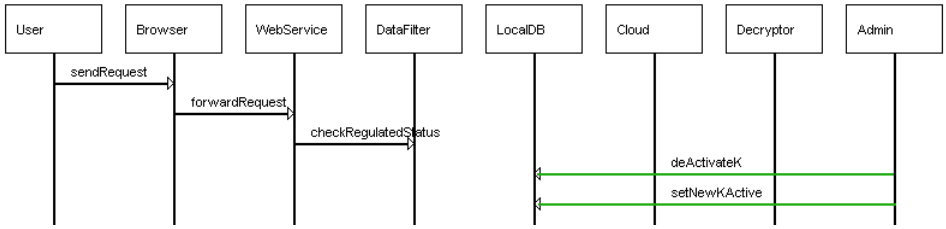
Table 6.9: Results from Case Study 2 for the composition of sets of scenarios using the SecArch. Scenario names are listed in Figure 6.7. The 'Maximal Traces' column represents the number of traces in the architecture model. These traces are inspected for race conditions. The 'Positive Scenarios' column represents the number of acceptable scenarios detected, and finally the 'Security IS' represents the possibly malicious behaviours detected. The term 'unique' represents the security threats that were not found in the IS approach (i.e. unique to the SecArch approach).

	Scenario No.				Maximal Traces	SecArch			
						TotalRace Conditions	Total ISs	Positive Scenarios	Security IS
Combo1	Scen10	Scen9	Scen1	Scen2	9	2	4	1	5
Combo2	Scen3	Scen6	Scen5	Scen1	13	22	4	22	2 (1 unique)
Combo3	Scen5	Scen4	Scen6	Scen10	9	4	3	2	3 (2 unique)
Combo4	Scen1	Scen11	Scen10	Scen6	16	17	9	20	4
Combo5	Scen3	Scen11	Scen6	Scen5	11	7	2	5	2 (1 unique)
Combo6	Scen4	Scen7	Scen5	Scen8	4	0	N/A	0	1 (0 unique)
Combo7	Scen9	Scen1	Scen10	Scen11	10	1	4	4	2 (1 unique)
				Totals	72	51	26	54	21 (14 unique)

For demonstration purposes, we will look at the results of scenarios in Combo1 (see Table 6.9). We have taken four scenarios and composted them together to generate the

architecture behaviour model of these scenarios. Initially, we had one positive IS (see Figure 6.7), and a total of nine traces of possible behaviours in the architecture behaviour model (these are the traces checked incrementally for race conditions). All traces were translated into an MSC form and evaluated individually for race conditions using Alur's algorithm. The results from the race condition checks from a security-testing perspective are as follows:

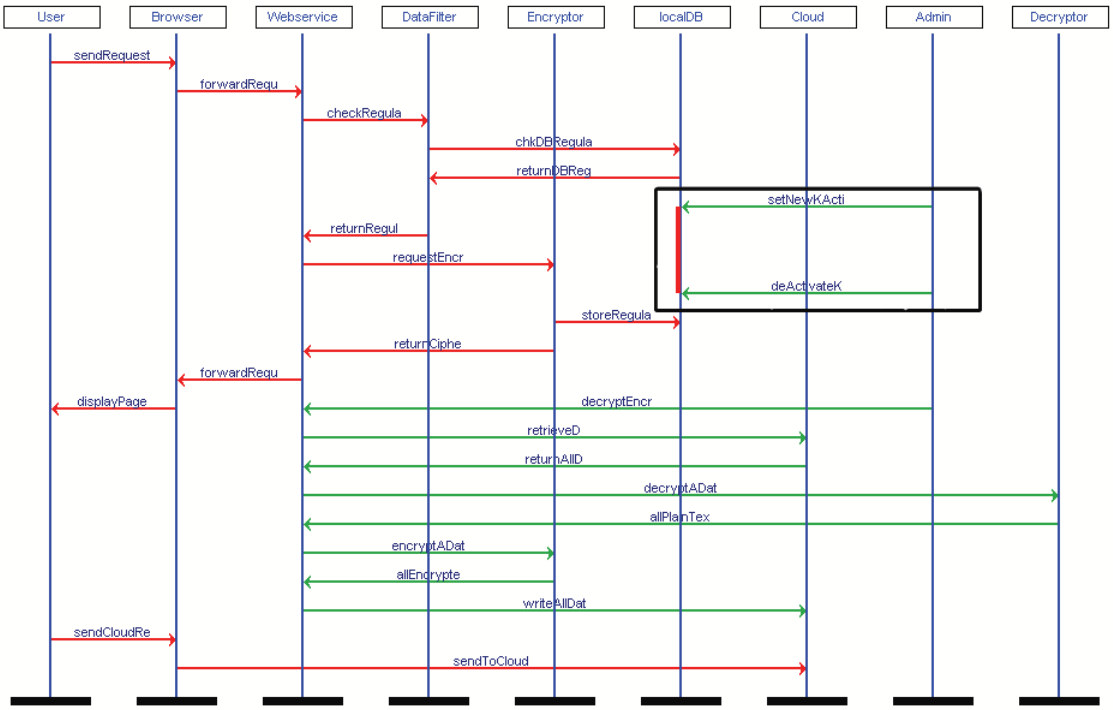
Figure 6.7 A Positive IS found due to the composition of scenarios 'User Registration' and 'Revoke Key'.



- Race Condition 1: A potential resource race condition in non-FIFO multi-queues when the administrator fires the two requests 'deActivateK' and 'setNewKActive'. When a new key is set as active, the compromised key remains active for the user to continue to use. Consequently, a user could read data from the cloud and attempt to re-encrypt it using the old key, and any new data re-written by the administrator (using the new key) will be overwritten by the new data submitted by the user. A subsequent functional problem to this is that when the user attempts to re-read the data, it will be decrypted using the active (new) key even though it was written using the old key, in which case the data would appear unreadable until the administrator was notified to decrypt it using the old, inactive key. This scenario is modelled in Figure 6.8. As the figure shows, traces derived from behaviour models represent a single global behaviour with respect to all scenarios involved in the interaction (ie, a composition of different scenarios taking place at the same time).

- Race Condition 2: The administrator is able to overwrite newly written data by the user. This could happen when the 'writeAllData' command arrives after 'sendTo-Cloud', resulting in data loss. This could be a common behaviour because encrypting

Figure 6.8 The composition of scenarios 'Revoke Key' and 'Save Regulated Data' indicates the presence of negative race conditions when the message 'setNewKeyActive' arrives after 'deActivateKey'.



and decrypting large amount of data could take a long time to complete. At the time of detecting this problem, there was no locking mechanism implemented to prevent such a problem since the architecture was not initially designed for multi-user access to the same data.

In order to investigate the negative consequences of these two race conditions, we have added these race conditions as possible scenarios and tested to see if their presence in the architecture would reveal another IS. The incremental test cycle detected four new ISs, described below:

- IS 1: Derived from the interleaving of *scen2: User registration* and *Race Condition 2* causing corruption of user data:

```
1: fetched data + invalid key + user saves first = corruption of user data
2: new data + invalid key + user saves first  = corruption of user data
```

- IS 2:When the user has sent a request to the web service to save data, the data is stored in a local database, and the encrypted version is returned back to the user as a confirmation of data security. Then it is the user's responsibility to update the cloud. The IS shows that the last two forwarded messages of data may not be executed; that is, the user may decide not to send the new data to the cloud even though they have already been stored in the *localDB* database. This results in data inconsistencies between the *localDB* database and the cloud.

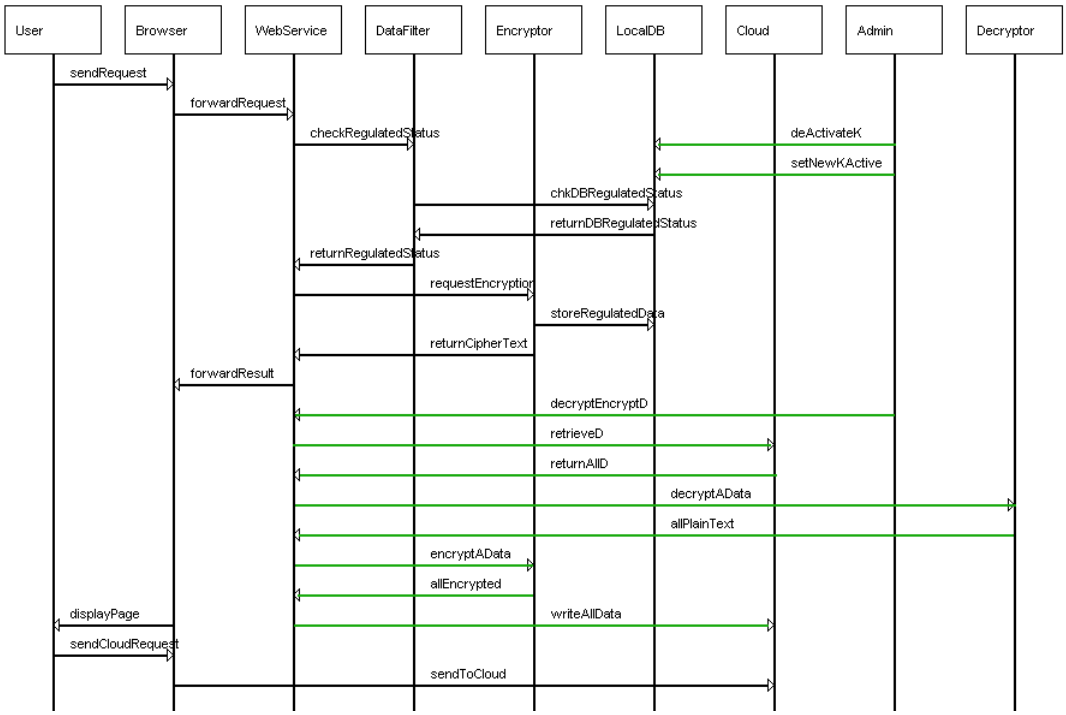
- IS 3: The web service halts in operation after all data is returned back from the cloud. Depending on how the system is implemented, this data may remain in the memory if no secure error handling mechanism is in place. An attacker could intentionally abuse the system to cause denial of service to the web service and to inspect the memory for resources left open (eg, data memory buffers).

- IS 4: A positive scenario that models an alternative behaviour overlooked in the design (see Figures 6.9). These scenarios were used to elaborate the behaviour model.

It is now possible to either continuously increment the search for race conditions since no more ISs are detected, or stop the search. The stopping criteria are left for the tester to decide upon based on timing or budget constraints. In this case study, we have only performed a single incremental cycle to explore the benefit of re-feeding race conditions into the LTSA-MSC for further IS testing. The same process took place for each combination of the five sets of scenarios, and results are presented in Table 6.9. We will summarise the results in the next Section.

Architecture Refinement We want to reflect on how the SecArch approach was able to identify key weaknesses that require refinement. These weaknesses are identified by the list of threats enumerated in Appendix A. We will look at three types of refinement triggered by SecArch to correct the current design:

Figure 6.9 A positive IS that models a valid composition of scenarios 'Revoke Key' and 'Save Regulated Data'



- *Preventing undesirable behaviours*: As scenarios reflected potential threats in their sequential form, the process of identifying which of the messages should not be allowed to execute in that threat scenario was a relatively a straightforward process. For example, deactivating a key and generating a new one should occur in a single transaction, where adding a new key should be sufficient to set the new key as the default (IS 1). Another oversight problem was that the administrator did not receive any acknowledgement of successful user revocations, which means that in some situations (eg, synchronisation), the database would not be available to accept the change (IS 13).

- *Adding new functionalities*: Other situations allowed us to identify new functionalities. For example, in the situation where synchronisation of the database and the cloud occurs, attempts to modify the database will fail. This would result in the information loss of any data or requests sent to the database. We need to implement a buffer that would execute as soon as the database becomes ready to receive data

(IS 21). In addition, an inconsistency between the database and the cloud seems possible in a number of situations, partially because the transaction for adding data to the local database does not require encryption for sensitive data, unlike the cloud, where a user is prompted with ciphertext to be committed to the database (ISs 4, 10, 12 and 18). Another important requirement is the security of the communication between the cloud and the web service; an eavesdropper is able to intercept communication that is sent in plaintext (IS 20).

- *Revising the access control model*: There is an implicit trust in the administrator in-house, where sensitive data is stored in plaintext in the database and all active and inactive keys are accessible (which can be dealt with by using third-parties to store keys). This violates the security 'principle of least privilege' [154], in which every process or user should operate using the fewest number of privileges necessary to complete the task. In our case, the administrator does not require access to user data to perform his designated tasks. The use of the SecArch approach detected a number of situations where an insider might violate confidentiality of the database (ISs 2, 9 and 19). There is also the possibility of a user maliciously changing sensitive fields to allow data transfer in plaintext (IS 20). In general, this calls for revision of user access roles and trust levels. We can observe that SecArch is capable of identifying situations that abuse the principle of least privilege because SecArch explores the potential interactions among privileged users/processes, so that improper uses of privileges are identified. A similar situation was identified in Case Study 1. In addition, it provided a rationale as to how and where we can refine the system by providing the testers with a precise description of the threat and its location.

Some of the above refinements were discussed with the developers [53], and the justifications we received were: (1) the original plan of the architecture was not designed for multi-user purpose, which meant weaknesses in the user access roles, as well as lack of thread-testing and synchronisation. (2) The exposure of ciphertext to users was for the

purpose of usability and sense of security assurance that the data is secured before being forwarded to the cloud.

Automation and Tool Support As Table 6.9 reflects, the number of race conditions detected were relatively large compared to security threats detected, which could mean that the effort required to check these conditions would be laborious. The reason behind the large number is that we needed to split the scenarios into multiple compositions, which meant that the race conditions were appearing repeatedly across the scenarios. This is avoidable using automation in parts of the SecArch process. As we have discussed in Chapter 3, security testing requires semi-automation, rather than full automation. Human involvement is needed to counteract the creative nature of hacking. Thus, some automation is required to reduce labour tasks, such as the avoidance of false positives and repeated scenarios that were previously detected, or tasks that require high computations, such as the detection of ISs.

To achieve semi-automation, the following steps can be used to reduce laborious work:

1. Alur's algorithm provides the detected traces of an MSC diagram in a textual form (eg, *Race 3 (type 2) deActivateKey* $<->$ *requestDecKey*, which means that the order of the two messages may alternate). The message semantics of the IS approach require that messages are unique in their functionality (ie, no two messages can have the same name with different functionalities), which allows the approach to synchronise on shared messages (with the same name). Thus, from all MSC diagrams, we can compose a list of unique race conditions that are not repeated across MSCs. This will omit inspections of previously tested traces.

2. Automate the translation of scenarios backward and forward from the LTSA-MSC (XML) used to detect ISs into UBET (MS common console document), which is used to detect race conditions. The translation requires a textual comparison between the two file formats.

6.3.2.5 Summarising the Findings of Case Study 2

In the beginning of the case study, we have stated three propositions where we mentioned that introducing the IS approach will (1) allow for the detection of threats in the architecture, and will (2) adapt to changes in agile-like developments. (3) Introducing SecArch as an enhancement version to the IS approach will improve the search-space of threats, and it will result in the number of threats detected using SecArch being greater than the number of those detected using the IS approach. In this section we will summarise the results of our findings with respect to these propositions:

- In all the stages we covered in this case study, we were able to detect security threats using the IS approach. In the first stage, the industrial-party reported one critical threat; in stage 2 we reported seven of 22 ISs that poses security threats, and in stage 3 we detected additional 21 threats.

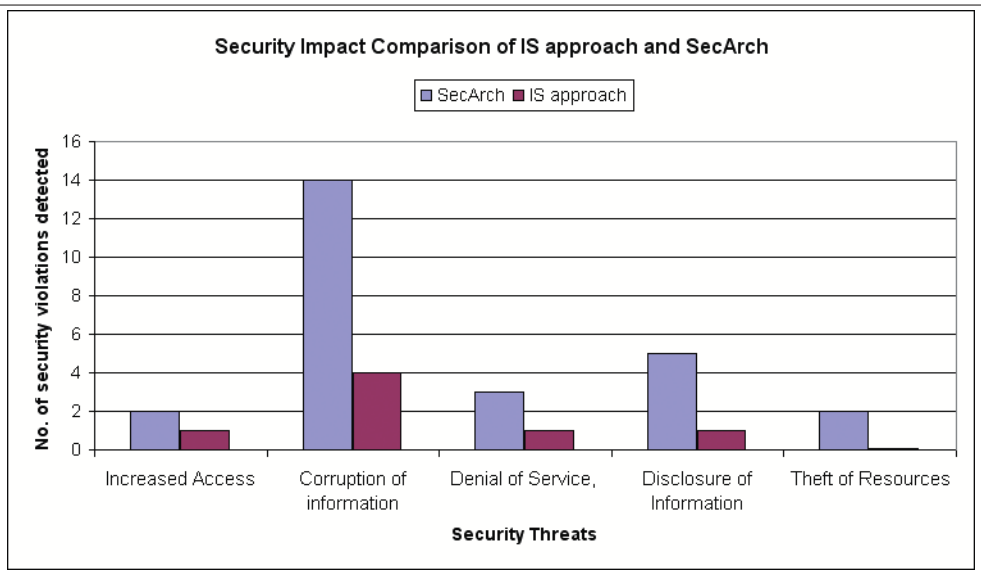
- To satisfy proposition two, for an approach to be consistent with agile principles, it must be able to adapt to changes incrementally as new requirements become available. We have seen demonstrations of this in the refinement in Section 6.3.2.4, where we made changes, added/removed functionality and the testing process continued. In this case study, we showed that we were able to reconstruct the architecture as the new agile iteration was invoked to add new functionalities. The results of the compositions presented in Table 6.6 show that we have detected a total of 22 IS scenarios, of which seven constitute Security IS threats. The table reflects a successful integration between the first iteration conducted by the industrial-party case study (shown in Table 6.6 as Combo6), and the rest of scenario combinations 1-7. These results indicate two aspects of testing:

    1. We were able to test the system without direct interaction with the first agile iteration (Combo6), as shown in the scenarios of Combo1 in Table 6.6. This would allow the testing of a coherent set of functionalities that may not interact with other sets of scenarios.

2. We have also tested different levels of interactions with the first iteration (as in combinations 3 and 4) in order to study the results of adding new functionalities into the system, and to investigate how the additions can break the security of the first tested iteration.

- In order to test the third proposition, we need to compare the SecArch results with the original IS approach. Tables 6.6 and 6.9 show that in the original approach, we were able to detect seven potential threats in the architecture model; whereas, using the SecArch approach, we were able to detect 21 threats in the architecture model (of which 14 threats where unique to the SecArch approach and were not detected before using the IS approach). The enhancement allowed us to explore various conditions that might occur due to a multi-queuing structure, such as the finding of race conditions in resources processes. With the presence of this information, we were able to study to see if the presence of race conditions would result in additional ISs. The demonstration in Section 6.3.2.4 showed an example of this: from two race conditions, we were able to identify four additional ISs. The same re-occurred for the other scenario combinations. The security testing implication of this is important as it allows us to study the effect of presence of malicious behaviour in the system, and how it can branch further to cause more damage. This is needed for situations in which refinement is not feasible. Without this enhancement, the additional 14 threats would have passed undetected. The graph in Figure 6.10 presents a comparison of the impact of the threats detected using the IS approach and the impact of those detected using the SecArch approach. The results indicates that: (1) the testing of SecArch is more significant than the IS approach in security violation impacts and, thus, that its incorporation into the original IS approach enhances the search space for the IS approach; and (2) that a combination of both approaches is better at detecting data-related threats (ie, corruption of information, disclosure of information).

Figure 6.10 Presents the impact of detected threats on the cloud architecture presented in Case Study 2 using the SecArch approach, and the IS approach. The graph shows that SecArch was able to detect more situations that were overlooked by the IS approach. The x-axis values come from the table presented in Appendix A.

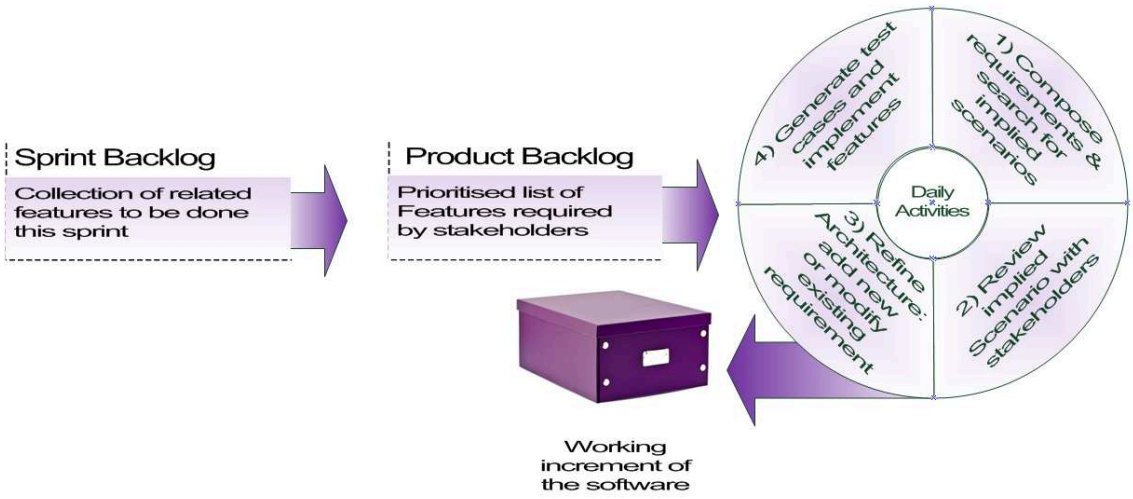


6.3.2.6 Discussion

The SecArch approach allowed us to detect the 14 additional security threats, which were design-specific behaviours to the architecture, in a systematic manner. expert judgement alone would have suffered from three weaknesses: (1) the fact that it is based on experience, which makes the results slightly biased, because the testers may be driven towards what they have previously encountered; (2) ISs are very difficult for humans to detect because they reflect what each component can visualise during runtime, and how that visualisation is composed with other components' limited vision; and (3) the detection of diverted and interleaving behaviours by humans is likely to be ad-hoc and prone to errors. SecArch addresses these limitations by drawing the focus onto the architecture's design-specific weaknesses that need to be addressed in order to build a trusted, secure architecture that will behave as intended. This automates the process of detecting ISs so that we can focus the testing process on what is inherently weak in the design, rather than basing the testing process on random estimations or trial and error. Following a systematic approach guarantees the detection of all possible and yet unforeseen behaviours (from a static view) that could affect the security of the system.

Figure 6.11 illustrates how the iterative process of our approach maps to SCRUM to produce a working increment of the developed software. The process starts with the collection of related requirements to be implemented, followed by the prioritisation of these requirements and then the composition of them incrementally (as done in the first agile iteration developed by the industrial-party) to produce the first working increment. The second iteration involved the addition of new requirements as the system evolved to produce the second working increment of the software. As we demonstrated, the testing process involved testing the iterations both individually and when integrated with each other in one increment. Notice that the type of documentation required is light-weight, and is essential to enumerate the requirements in scaled projects.

Figure 6.11 Mapping between the IS detection approach with SCRUM. This mapping shows how the steps of the IS approach satisfies the agile principles by promoting continues evolution.



6.3.3 Case Study 3: Distributed Smart Camera

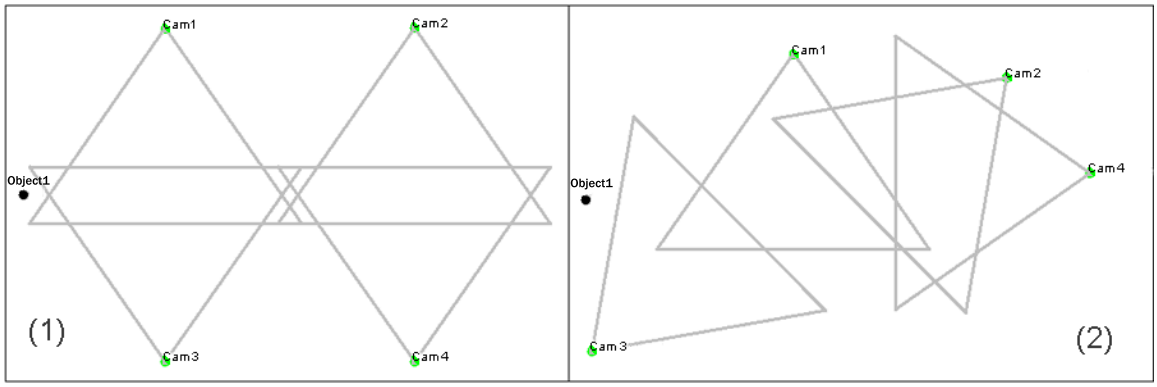
In this case study, we proposition that the introduction of the IS approach not only will it (1) allow for the detection of threats in the architecture, and (2) guide the refinement to produce a more secure architecture, but it will also (3) guide the selection of architecture alternatives and candidates with respect to the number of malicious behaviours detected on each candidate architecture. The presence of ISs indicates potential vulnerabilities that

act as a back door for architecture abuse; thus, we base the selection of architecture on the smallest number of threats detected. The method protocol, data collection and questions are discussed in Section 6.2.2. We will first introduce the architectures of the Smart Camera case study. We have chosen this case study for the purpose of: (1) testing the approach's applicability in high-dynamic situations with an increased level of uncertainty, where we have unpredictable and continuous change in topology and data structures; and (2) studying the approach when the scenarios were not present upfront (ie, only the implementation). This allows us to compare the approach's resilience in regards to dealing with various situations and any potential difficulties that may need to be addressed to enhance the approach. Two members have tested the architecture, one with security testing background, and one with architecture and testing background.

6.3.3.1 Introducing the Distributed Smart Camera

This case study is for a smart camera network developed by the EPICS research project [51]. The network consists of wireless sensor cameras working in a distributed configuration. Their aim of the project is to manage the tracking of objects with respect to the available camera resources. It consists of twenty-one Java files with 7695 total number of lines. The cameras start with limited knowledge about the network, such as the number of cameras, and improve their knowledge through message exchanges. This addresses the adaptivity of distributed system overtime as opposed to the acquisition of full knowledge about the cameras upfront. The communication between the cameras uses a passive multicast approach by sending advertisement messages of their objects to cameras only when an object is about to leave the field of view (FOV) of its current owner, and sending messages only to neighbouring cameras. In choosing this case study, we wanted to study with the following: (1) the dynamic and unpredictable environment, where modes of interaction cannot be predicted (ie, highly dynamic mode/composition), because the static approaches are unfit to evaluate the possible threats caused by the dynamic composition; (2) the timeliness and applicability of the IS approach in regards to a real life scenario;

Figure 6.12 Two architecture configurations for Distributed Smart Camera Case Study 3. The green circles represent the positions of four cameras in each of the architectures, the triangles dispersing from the cameras are the FOVs and the black dots are the moving objects.



and (3) the scale and dynamic trade-offs in which security is the primary concern.

The two architecture configurations we have tested are shown in Figure 6.12. The green circles represent the camera position in the view graph, the triangles dispersing from the cameras are the fields of view (FOVs) and the black dots are the moving objects. These architectures were modelled using the EPICS simulation package to allow us to check the accuracy of camera behaviours. We created ten simulation files, five for each architecture configuration, to represent the position and directional movement of the tracked objects, as shown by the red dotted arrows in Figure 6.13. The neighbouring relationships between the cameras in both architectures are shown in Table 6.10. Note that there is no relationship between Architecture1 and Architecture2.

Table 6.10: Neighbouring relationship between cameras

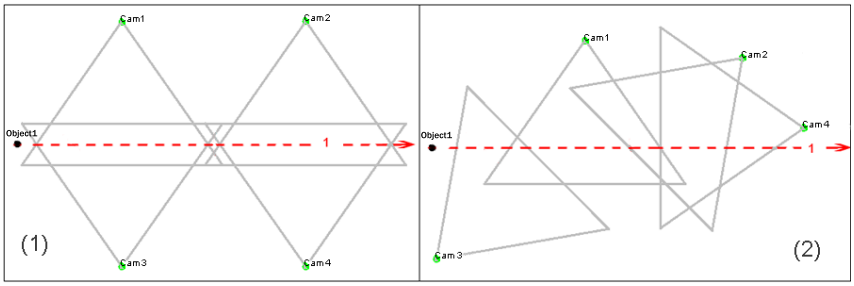
Cameras	Architecture1	Architecture2
1	2	2,3
2	1	1,4
3	4	1
4	3	2

6.3.3.2 Modelling the Distributed Smart Camera

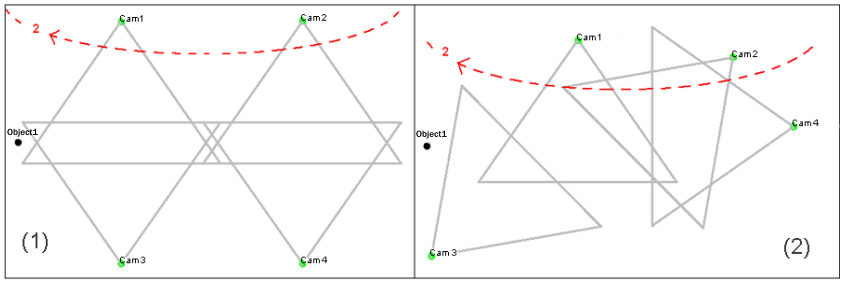
The number of scenarios in the architectures presented in Table 6.11 are extracted using automated Visual Paradigm tool [135]. We have experienced a number of limitations when

Figure 6.13 Five scenarios to be tested on two architecture configurations for Case Study 3.

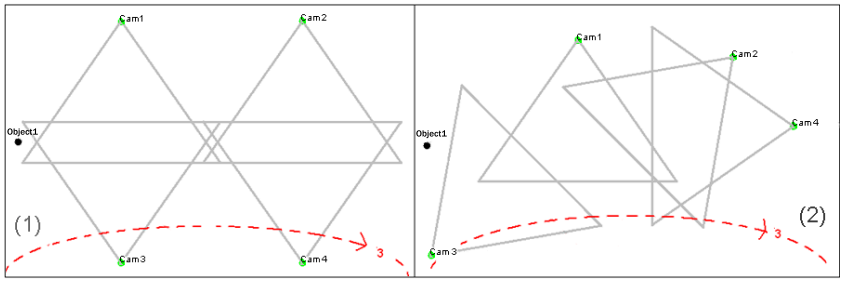
(a) Scenario1: Object1 moving from left to right (following the red dotted arrow), crossing the views of all four cameras in Architecture1 and Architecture2.



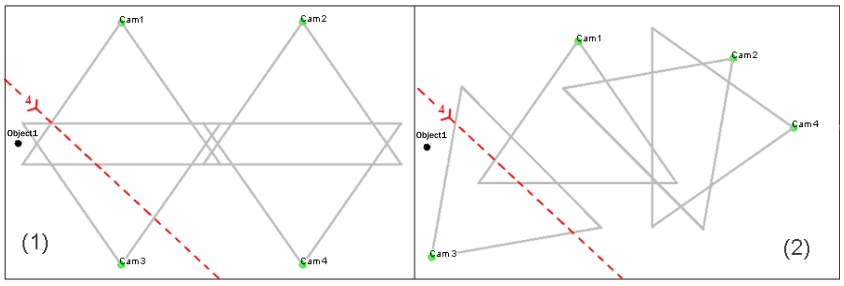
(b) Scenario2: Object1 moving from right to left, crossing the views of cameras Cam1 and Cam2 in Architecture1, and cameras Cam1, Cam2, and Cam4 in Architecture2.



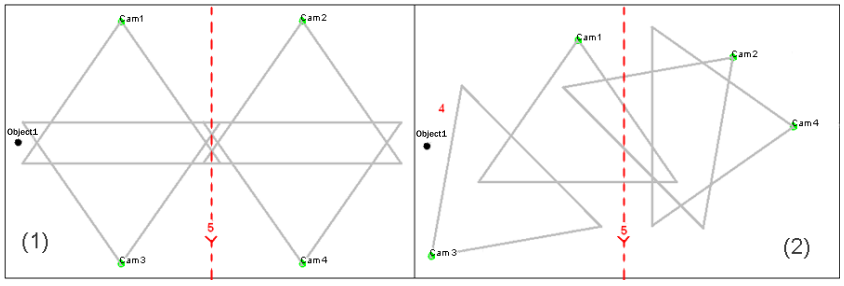
(c) Scenario3: Object1 moving from left to right, crossing the views of cameras Cam3 and Cam4 in Architecture1, and cameras Cam2, Cam3, and Cam4 in Architecture2.



(d) Scenario4: Object1 moving from left downwards, crossing the views of cameras Cam1 and Cam3 in Architecture1 and Architecture2.



(e) Scenario5: Object1 moving from top to bottom, crossing the views of all cameras in Architecture1, and crossing cameras Cam1, Cam2, and Cam3 in Architecture2.



modelling the scenarios using the IS approach, these limitations were directly associated with the LTSA-MSC tool for detecting ISs in the architecture models:

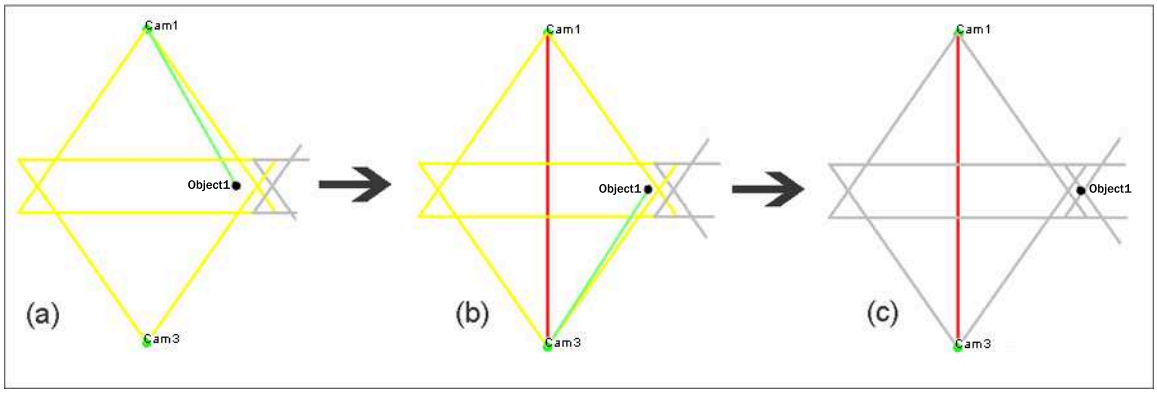
Table 6.11: Scenarios elicited for Case Study 3: Distributed Smart Camera

	Scenario	Description
1	Ask For Confidence	Confidence value represents how much of the object can be seen by the camera. The higher it is the stronger the view.
2	Start Tracking Of Object	Indicates that the camera acquired possession over the object
3	Start Search	cameras send messages to neighbours to find an object
4	Stop Search	cameras can send messages to neighbours when an object is found to stop them from searching and exhausting resources
5	Losing Object	Indicates that the object is about to lose the FOV of the camera
6	Ask For Confidence	Cameras ask neighbouring cameras about their view of the object
7	Camera Start Tracking	Cameras pass possession of objects to other cameras
8	Object Found	Indicates that objects have entered the FOV of some camera
9	Remove Tracking	Internal camera call to remove an object when when the camera stops tracking it

- Many camera moves relied on the internal state of its data structures, such as the list of neighbours and the list of *advertisedTo* cameras (to whom the object was advertised), which meant that we needed to determine the sequence of the message calls based on our awareness of the neighbouring tables and the position of the object. The difficulty was in interpreting the state of the camera in order to make the correct move, or at least to see potential alternatives in the data structure.

- There were difficulties in modelling when the camera made a decision based on a value returned (eg, best confidence value). Consider the following scenario in Figure 6.15 (the behaviour is modelled in Figure 6.14): when Object1 enters the view of a camera, the FOV turns yellow (ie, for both Camera1 and 3), and the camera tracking the object is linked with the object in green (ie, Camera1). As the object is leaving the view of both Camera1 and Camera3 with the same confidence levels, Object1 should not have been granted to Camera3 (notice the red link between Camera1 and Camera3 indicating the pass of object), because its confidence level is not better. It means there is unnecessary message exchange and passage of Object1 because Camera3 immediately lost track of Object1 the moment it received it.

Figure 6.14 Simulation movement of Object1 (black circle) in the FOV (yellow triangle) of cameras Cam1 and Cam3. In figure (a), the object is seen by both cameras, but it is tracked by Cam1 (represented by a green connector). In figure (b), the Cam3 starts tracking the object as the objects moves further, and the neighbouring knowledge between Cam1 and Cam3 is acknowledges in a red connector. In figure (c), the object leaves the view of both cameras and neither keeps tracking the objects, however, the awareness of neighbouring relationship remains valid.



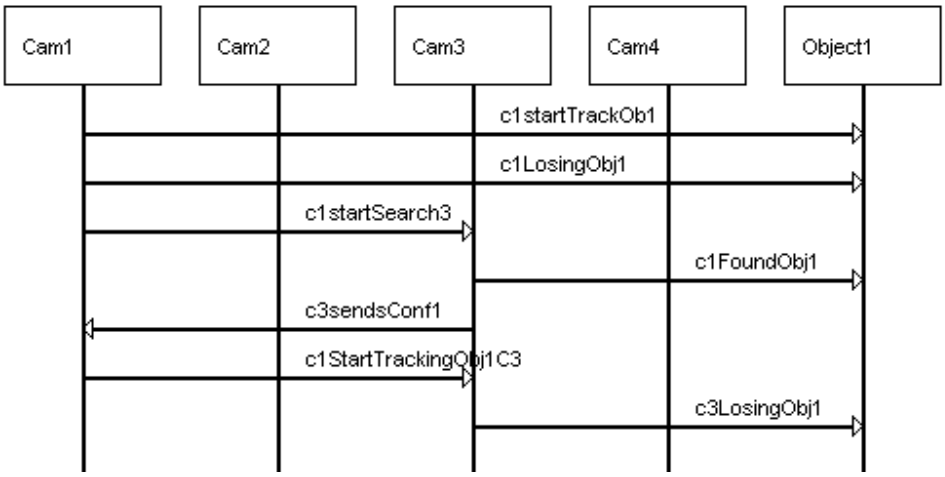
- Because objects move, we cannot simulate the position of the objects, so we would be unable to detect if a camera is falsely claiming a view of an object automatically.
- We could not model timing *duration* (ie, if an action takes place in a certain period of time). If we wanted to model a camera that issued a message or made a move after 10 seconds, we would not be able to do so using the limited LTSA-MSC interface.

The above limitations open up future research into how we can further enhance the search space of ISs to include data structures, internal behaviour and modelling time-related behaviour. For example, an IS might occur if the component did not advertise to all its *advertisedTo* cameras, or it may accept the release of an object to a message that was initiated outside of its neighbouring relationship.

6.3.3.3 Classifying ISs

Using the LTSA-MSC tool, the total number of ISs detected in Architecture1 are 16, and 36 in Architecture2. The classification process for identifying Security ISs was carried out by two testers. The method protocol for the classification process is explained in Section 6.2.2. From the total number of ISs in Architecture1, we found two Security ISs with five

Figure 6.15 A scenario modelling the message communication between cameras Cam1 and Cam3. As Cam1 loses the view of Object1, it communicates with its neighbouring cameras (namely Cam3) to start searching for Object1. Because Cam3 acknowledges that it can see Object1, Cam1 passes the tracking of Object1 to Cam3.



potential threats, and we found four Security ISs in Architecture2 with eight potential threats. These Security ISs with their threats are explained in Table 6.12. Reflecting on the taxonomy [70], the types of attackers likely to attempt breaking into the system are:

- professional criminals - attack computers for personal financial gain.
- hackers - attack computers for the challenge, the status or the thrill of obtaining access.
- vandals - attack computers to cause damage.
- voyeurs - attack computers for the thrill of obtaining sensitive information.
- spies - attack computers for information to be used for political gain.
- corporate raiders - employees (attackers) who attack competitors' computers for financial gain.

The results of the classification are presented in Table 6.12. Identifying the source of the attackers helped in visualising the different ways in which the system can be abused. However, situations such as IS3 –and one threat in IS4– were caused by incorrect behaviour, and so were not necessarily initiated by an attacker. Having explored how each

attacker might want to attack the system, it was not clear how an attacker might benefit; for example, from two cameras tracking a single object. This does affect the availability of the cameras and thus might lead to denial of service when the cameras fail, but the attackers did not initiate such a process. This backs up the motivation of this research where we stated that security could be violated by functional composition; thus, focus on composition should be explored further in the research community. Another limitation we found in classifying the results (ie, 'Increased Access, Disclosure of information, Corruption of information, Denial of service, and Theft of resources') of IS4, in which the problem was related to one camera giving up ownership of an object, believing it had successfully passed the track to another camera. The closest option is 'Denial of Service' because the cameras cannot perform as expected, though in the security community this term is often used when an attacker causes such denial of service, and we have said that the threat in IS4 was not caused by an attacker.

6.3.3.4 Security ISs Results

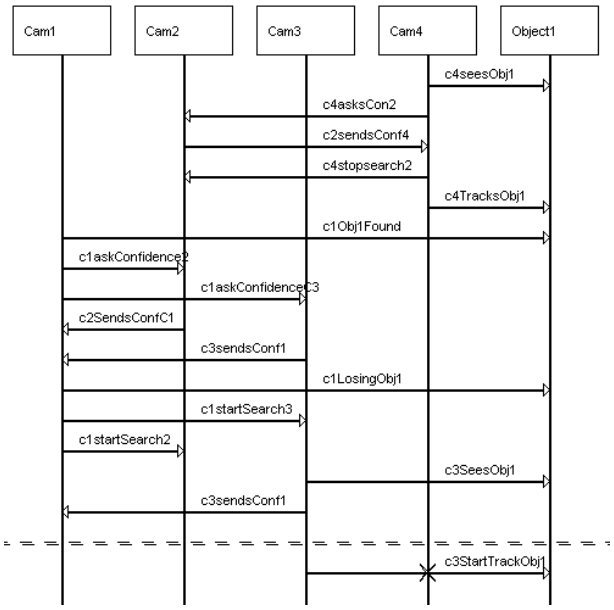
Consider the following IS (shown in Figure 6.16) that occurs in Architecture2 between Camera1 and Camera3. The IS composition indicates that Camera3 grants itself possession over Object1 before it receives responses from other cameras. This would be a valid move in a situation in which time had expired or a camera had run out of power. However, because we are not modelling this possibility, the detection is not desirable. This has a number of security impacts on the architecture:

1. Malicious cameras could grant themselves access over objects to block other cameras from tracking them. This can be achieved by the cameras claiming very high confidence values. A behaviour of this type can be abused in many ways depending on the type of system that implements the configurations; for example, a malicious user may leave unattended luggage in an airport and prevent other cameras from tracking to see if the luggage is picked up afterwards.

Table 6.12: Classification results of detected ISs case study 3

	IS	Attacker	Target	Results	Security Outcome
Arch2	IS1	Professional criminal, voyers, vandals, hackers, corporate raiders	Objects	Increased Access, Denial of Service, Corruption of Information	Malicious cameras can grant themselves access over certain objects to block other cameras from watching over the objects being tracked, this can be done by claiming very high confidence values.
		Spy, hacker, voyers, corporate raiders	Network topology	Increased Access, Disclosure of Information	A malicious camera can also intercept the network and reveal everything about the communication and topology configuration
		Professional criminal, voyers, vandals, hackers, corporate raiders	Cameras	Corruption of Information Denial of Service,	Sends false information to disrupt the cameras or waste their energy
		Professional criminal, vandals,	Cameras	Denial of Service, Theft of resource	Camera1 might go offline maliciously, possibly by forced removal, or physical damage.
	IS4	N/A	Object	Denial of Service	Camera2 gave up observation of object1 on the believe that Camera1 started tracking the object, thus object1 is no longer trackable (until another camera finds it).
		Professional criminal, vandals, corporate raiders	Cameras	Corruption of Information Denial of service	Camera4 did not respond, it could have been brought down by removal or damage, or the message might have gotten lost due to intentional inference. An attacker can set up an antenna near by the camera to distract the signal strength.
	IS5	Professional criminal, vandals, corporate raiders	Camera	Corruption of Information Denial of Service	Two neighbouring cameras may track the same and exhaust the network. When Camera2 received object tracking of Object1, Camera1 had already begun tracking it, thus the stop search is not expected at this stage. This is caused by delay in Camera4's response to Camera2's confidence value. Message latency could also be caused by malicious inferance caused by wireless signal jammers since wireless cameras transmitters use one of three standard ranges for transmission: 900MHz, 1.2GHz, and 2.4GHz.
	IS6	Professional criminal, voyers, vandals, hackers, corporate raiders	Camera	Increased Access Denial of Service, Corruption of Information	Indicates that Camera3 is claiming to see Object1 when it is outside of its view. It is possible for malicious cameras to claim they can see an object and are tracking it even if the information is not valid.
Arch1	IS2	Professional criminal, voyers, vandals, hackers, corporate raiders	Cameras	Corruption of Information Denial of Service,	Allows a malicious object to exhaust the cameras by continuously asking cameras to check their confidence, not only will this cause denial of service, it can bring the camera down due to exhaustive computations of the confidence value.
		Professional criminal, voyers, vandals, hackers, corporate raiders	Objects	Increased Access, Denial of Service, Corruption of Information	Malicious cameras can grant themselves access over certain objects to block other cameras from watching over the objects being tracked, this can be done by claiming very high confidence values.
		Spy, hacker, voyers, corporate raiders	Network topology	Increased Access, Disclosure of Information	A malicious camera can also intercept the network and reveal everything about the communication and topology configuration
		Professional criminal, vandals,	Cameras	Denial of Service, Theft of resource	Camera2 might go offline maliciously, possibly by forced removal, or physical damage.
	IS3	N/A	Cameras	Denial of Service,	Two non-neighbouring cameras tracking the same object

Figure 6.16 IS detected in Architecture2 indicating possible misbehaviour in Camera3, where Camera3 tracks Object1 without waiting to receive the confidence value from its neighbouring cameras.



2. A malicious camera could intercept the network and reveal information about the communication and topology configuration. This can reveal confidential footage, how a company operates, who is being tracked, etc. The code given in this study did not include encryption and, according to the developers, the system was not designed with security in mind.

3. A malicious camera could send false information to disrupt the cameras or to waste their energy; for example, asking continuously for the computation of confidence. There is no guard against such attacks in the code and no authentication of cameras joining the field. The assumption is that all cameras in the field are trustworthy.

4. In this IS, Camera1 might have maliciously gone offline, possibly by forced removal or damage that prevented it from responding back to Camera3.

The results of the other 5 ISs detected are as follows:

- Architecture2 IS4: Camera1 received a tracking token even though it lost view of the object. Camera4 did not respond although it had a better view of the object.

This IS has two implications: (1) Camera2 gave up ownership of the object based on the belief that Camera1 started tracking the object, and thus Object1 is no longer trackable (until another camera finds it); and (2) Camera4 did not respond. It could be out of reach, have run out of power, have been brought down by removal or damage or the message could have been lost. In the latter situation, the problem could be related to reliability –or it could be interference related– where an attacker sets up an antenna near the camera to distract the signal strength.

- Architecture2 IS5: Two neighbouring cameras may track the same object. When Camera2 received tracking of Object1, Camera1 had already begun tracking it; thus, the 'Stop Search' command is not expected at this stage. Observing the situation, Camera4 did not respond in a timely manner to Camera2's confidence value. Thus, Camera1 had enough time to track the object. Message latency is common in distributed networks; however, it could also be caused by malicious interferences caused by wireless signal jammers since wireless camera transmitters use one of three standard ranges for transmission: 900MHz, 1.2GHz and 2.4GHz.

- Architecture2 IS6: Indicates that Camera3 is claiming to see Object1 when it is outside of its view. If we consult the configuration of Architecture2, we see that if Camera1 had passed the object to Camera 2, Camera3 could not possibly see Object1 because Object1 would be moving towards Camera4 and, thus, is far from Camera3. It is possible for a malicious camera to claim that it can see an object and that it is tracking the object even if this information is incorrect. The damage from this may not be limited to a single object; it can propagate to prevent other cameras from tracking any trackable object in the field.

- Architecture1 IS2: In the situation in which the object was moving from Camera1 and Camera2 continuously, both cameras kept alternately finding and losing the object. The problem occurred when Camera1 lost track of the object and, when it found the object again, tracked it immediately. The correct behaviour requires

that Camera1 ask for confidence first, and if other cameras have less confidence in their views of the object, only then can Camera1 track the object. This threat is similar to the IS1 found in Architecture2, but both threats occur for different cameras with different neighbouring relationships. This configuration also allows a malicious camera to exhaust the other cameras by continuously asking components to check their confidence. Not only will this cause denial of service, but it can also bring the camera down due to exhaustive computations of the confidence value.

- Architecture1 IS3: Both Camera1 and Camera3 started tracking Object1 because they are not neighbouring cameras in their visual graph. The awareness of neighbouring cameras can improve over time, but with the current limitations, IS detection does not take internal states into account, which means this IS will likely occur in the beginning of the configuration runs before the cameras recognise all of their neighbours. However, this may not be the case if the cameras are distant from each other. This will present an interesting trade-off between levels of security and utility, whereby security specialists may require the high visibility of an object (eg, to capture someone's facial features on more than one camera), and thus accept the IS as positive, or care more for the availability of the cameras for a longer period of time since wireless networks suffer from power management issues. This trade-off happens between integrity and availability as part of the CIA security properties and can only be determined by the type of critical security level required. A related trade-off will take place at the level of the network-learning algorithm, which may be optimised more for utility than security. In such a case, the IS will be negative because it exhausts the energy of two cameras. In future research projects, it will be interesting to see how security can guide the network-learning algorithm for optimised security and utility.

In total, there were 14 positive scenarios detected in Architecture2, and 32 in Architecture1. These positive scenarios were used to confirm the accuracy of cameras' behaviours across the architectures. We believe that Architecture1 has more ISs (including posi-

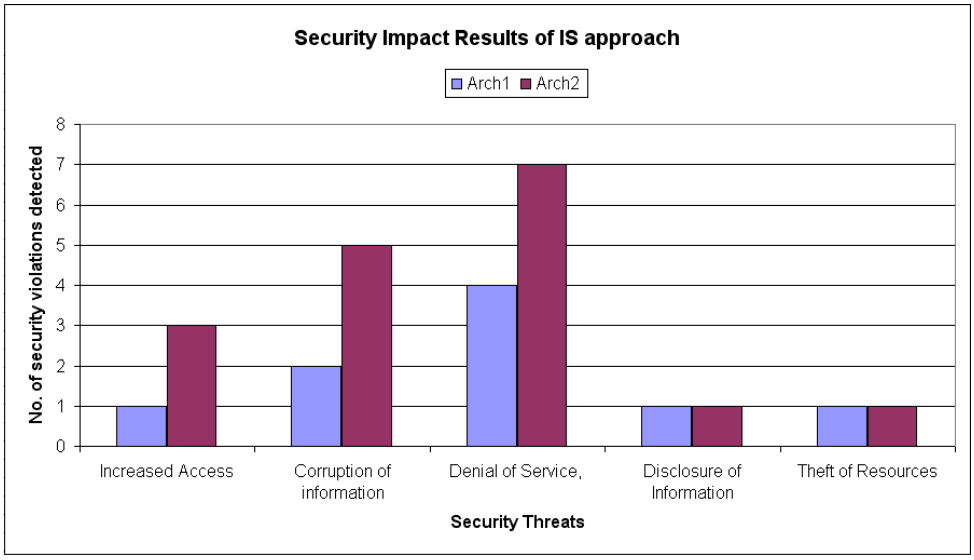
tive ISs) because the number of interactions across its cameras is less than Architecture2 due to the cameras' neighbourhood configurations. The less knowledge the cameras have about their surroundings, the more likely it is that the specifications will contain gaps, and will require higher synchronisation between components.

6.3.3.5 Architecture Selection

One of the propositions we have set for this case study is that the IS approach will guide the selection of architecture alternatives and candidates with respect to security. We have also stated that the number of security threats found in each architecture determines which of the candidate architectures is more secure than the others. From the detected negative scenarios, we found eight potential threats in Architecture2, and five in Architecture1. The difference in the number of threats detected is not significant. However, in terms of the potential impact of these threats, the difference is relatively visible, as illustrated in Figure 6.17. For example, Architecture2 is almost twice as likely to experience Denial of Service attack, and has over 70% chance of Corruption of information, compared with 28% chance of Corruption of information in Architecture1. It also has thrice as likely to experience Increased access. Both architectures have equal chance of Disclosure of Information and Theft of resources. As a results, Architecture1 is selected as the least vulnerable architecture.

General Observation: Consider the visual configurations in Architecture1 and Architecture2 shown in Figure 6.12. The visual configuration indicates that Architecture2 offers better coverage in terms of tracking possibly malicious objects. However, the conclusion we have arrived at with respect to the IS detected is that Architecture1 is likely to experience less security-related threats. This observation indicates the limitations in using expert judgement alone when testing distributed systems or multi-threaded systems. This is because expert judgement is likely to make judgements regarding the visual configurations/scenarios that are observable at the static level, as opposed to the real dynamic behaviour (which is harder to visualise from the top-level view) that is likely to be the

Figure 6.17 Graph showing the difference between the number of detected threats in architectures Arch1 and Arch2. The results indicate that the threats found in Arch2 are more significant than those found in Arch1. These numbers are collected from Table 6.12 column 'Results'



source of security problems. However, human judgement can be used to elaborate the results further, such as by taking into account the fact that the cameras might be blocked by a physical barrier, which is one threat that the IS approach cannot detect because it requires the modelling of foreign objects outside of the functional scenarios. We involve expert judgement only on detected ISs, and how that specific trace could lead to threats. We have chosen to only include threats that are detectable by the IS approach to test the IS approach's ability to detect threats, and to strengthen the confidence in its fitness regarding the detection of security threats.

6.3.3.6 Architecture Refinement

We have propositioned that the introduction of IS approach will guide the refinement to produce a more secure architecture. We have also highlighted a number of conditions to measure the satisfaction of this proposition in Section 6.2.2. In this section we will review the introduced refinements to address the detected threats presented in Section 6.3.3.4.

- Architecture1 IS2: The cause of this threat is that cameras can take ownership of objects when other cameras do not respond. This situation is acceptable when cameras die or are removed from the network, but in other circumstances it could be malicious. To address this problem partially, it could be required that cameras should always reply when they cannot see an object by sending '0' confidence. However, this will add a communication overhead and will not target the catching of malicious cameras. Let us explore how we can target each threat:

    1. Malicious cameras can grant themselves access to certain objects to block other cameras from watching the objects being tracked. This can be done by claiming very high confidence values. The refinement should include a trusted component that keeps a list of authorised cameras such that when cameras join the network, they register themselves, possibly by presenting a valid cryptographic signature. The registered keeper can announce the new camera by assigning an identifier to it, which will be used for any future communication with other cameras. This would guard against malicious cameras sending signals to authorised cameras. However, this may not guard against a trusted camera that was hacked by a malicious attacker. In practice, problems of this nature are addressed using intrusion detection by monitoring the behaviour of all of the cameras and determining abnormal behaviours.
    2. A malicious camera can also intercept the network and reveal information about the communication and topology configuration. All communication should be done via encrypted channels.
    3. Sends false information to disrupt the cameras or to waste their energy. The above refinements should prevent this threat from occurring.
    4. Camera1 might go offline maliciously, possibly by forced removal, or by physical damage. This would require physical protection.

- Architecture1 IS3: Two non-neighbouring cameras may track the same object and exhaust the network. This could be handled by requiring that objects send an announcement of every object they start tracking to the entire network to ensure that all cameras are aware of which is tracking what. Although this exhausts the network, the cost of two cameras (or more) tracking the same object is greater (power-wise). A refinement of this type is architecture-configuration-dependant, and would require statistically calculating the likelihood of two cameras tracking the same object against the cost of continuous broadcasting of messages. This raises the trade-off problem discussed earlier, where we either increase communication to enhance utility and security as a whole, or reduce communication to enhance availability (recall that availability is a security property).

- Architecture2 IS1: Refinement options for threats 1-4 have been addressed in Architecture2 IS2, as they are of a similar nature.

- Architecture2 IS4:

    1. Camera2 gave up observation of Object1 based on the belief that Camera1 started tracking the object; thus, Object1 is no longer trackable (until another camera finds it). An acknowledgement should be sent to confirm the successful migration of ownership. When trading-off between communication and utility, this message may be dropped in favour of more utility over communication. However, in practice this will not offer more utility because cameras will lose track of objects (ie, waste of utility). At the security level, the impact of losing track of objects is more costly than utilising power to send an acknowledgement message upon every swap of ownership.
    2. An attacker can set up an antenna near the camera to distract the signal strength. This would require physical protection.

- Architecture2 IS5: Two neighbouring cameras may track the same object and exhaust the network. This could be handled by requiring that objects send announce-

ments about every object they start tracking to their neighbours. However, this would not guard against IS3 detected in Architecture1. Thus, we adopt the solution presented for IS3 to solve both neighbouring and non-neighbouring conflicts.

- Architecture2 IS6: Indicates that Camera3 is claiming to see Object1 when it is outside of its view. This is very difficult to detect, and requires intrusion detection by monitoring the behaviour of all cameras to determine its occurrence.

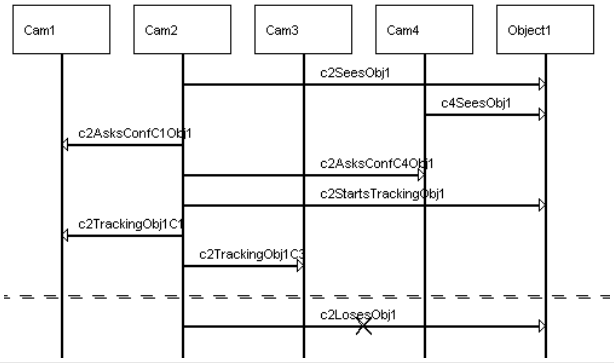
Refinements such as encryption channels are difficult to model using the IS approach. However, we have studied the refinements for IS 3, 4(1) and 5 to confirm the security status of the architectures after the refinement. The results are as follows:

For Architecture2 with the Higher Security Impact:

- For four cameras with five scenarios, 31 message overheads were added for refinement. These were distributed across the five scenarios. The behaviour of Scenario 3 had been modified to reflect the new behaviour of the cameras with the new information provided to them.

- The number of ISs found in Architecture2 dropped significantly, from 14 to 6. The number of Security ISs compared to the overall total of eight threats, is one, as shown in Figure 6.18. This IS indicates that Camera2 can send acknowledgement only to cameras 1 and 3. This may happen if the camera is removed or stopped, the message is lost or corrupted, or Camera4 is under attack through the use of antennas or signal jammers to distract signals from reaching the cameras. A future refinement will be needed to deal with potential message losses, whether intentional or unintentional. This satisfies one of the measurements of the proposition, where we highlighted that: 'If the number of threats detected after the refinement is less than the number detected initially, then the IS approach produced a more secure architecture'.

Figure 6.18 Detected negative scenario found in the refined version of Architecture2. The scenario indicates that camera Cam2 did not inform its neighbouring camera Cam4 about its process of tracking Object1.



For Architecture1

- For four cameras with five scenarios, 32 message overheads were added for refinement.

- The number of ISs found in Architecture1 had dropped from 32 to 21. We found two Security ISs: (1) Requesting that cameras announce their tracking of objects did not always protect against two cameras tracking the same object, because when two cameras see the same object at the same time, by the time they make the announcements, they have already begun tracking it. This is hard to guard against because each camera is allowed to make its own judgement given the information it has. However, a future refinement should involve the adding of a new requirement in which these two objects negotiate which should own the object (based on the confidence value). (2) If we only send an announcement when a camera owns an object, and if we do not send an announcement that the object is about to be lost to all cameras, then once the cameras find the lost object they will assume that it is still owned by the camera that sent the announcement. Thus, further refinements are needed to send an announcement to all cameras when an object is losing its view of an object. To summarise, in total we had five potential threats in this architecture, we now only have two threats; this satisfies one of the measurements of the proposition, where we highlighted that: 'If the number of threats detected

after the refinement is less than the number detected initially, then the IS approach produced a more secure architecture'.

6.3.3.7 Summarising the Findings of Case Study 3

In this study, we made three propositions that the introduction of the IS approach will (1) allow for the detection of threats in the architecture, (2) guide the refinement to produce a more secure architecture, and will (3) guide the selection of candidate architectures that is less vulnerable to threats. To briefly summarise the results:

- We have detected 16 total number of ISs in Architecture1, and 36 in Architecture2. Of the 16 ISs found in Architecture1, two are classified as Security ISs for which five threats stem from these behaviours. Of the 36 ISs found in Architecture2, we found four Security ISs for which eight threats stem from these behaviours. Figure 6.17 shows a graph demonstrating the security impact of these threats in comparison.

- We have mentioned in section 6.2.2 that the satisfaction of the second proposition is measured by the number of Security ISs detected before and after the refinement. For Architecture1, we had initially found 5 threats before the refinements, and found 2 threats after the refinements. For Architecture2, we had found eight threats before the refinements, and 1 threat after the refinement. This concludes that we have satisfied one of the criteria of the proposition: 'If the number of threats detected after the refinement is less than the number detected initially, then the IS approach produced a more secure architecture'.

- From the above results, we have concluded that Architecture1 is more secure than Architecture2 because it is less vulnerable to threats, with lower security impact.

The conclusion we make is that, the propositions we have set in the initial case study have all been satisfied, and that from these preliminary results provided us with very useful insight into the security posture of the application and, in particular, that the IS approach

seemed to be well-suited for the detection of potentially corrupt information. The case demonstrated a novel application of the IS approach in distributed smart camera system; we saw how the IS approach was used to detect malicious cases of utility manipulation in which a camera was able to prevent objects from being handed over to other cameras and, as a result, prevented other cameras from tracking objects. These malicious behaviours significantly impact the utility of the camera system. We studied the refinement of the architectures based on securing the detected IS threats, and found that the IS approach detected a number of trade-offs that were used in the refinement decision process. We also found situations in which the refinements were anticipated to be secure but, with further studying, the results did not improve the security posture; additional refinements are needed to secure the architecture.

6.4 Summary

Testing for security vulnerabilities is complicated due to the fact that they often exist in hard-to-reach states and appear in unusual circumstances. The analogy that the IS approach makes with ISs is simple, yet powerful enough to provide a basis for the analysation of the architecture's security posture. We have used this analogy to address the detection of any hidden behaviours in the architectures. The IS approach builds on a sound approach for identifying ISs [185]. The IS approach's security detection abilities have been confirmed through different case studies, as was presented in Section 6.3. The conclusions we made from these case studies are that: (1) The approach is capable of detecting security threats, even in non-closed environments such as the cloud. The approach takes into account the incompleteness of the architecture model to present a realistic representation of multi-threaded systems. (2) The approach is adaptive to changes that might take place in the architecture, which allows us to retest the architecture's security posture as changes develop. This provided insight into whether or not changes should be committed. To further confirm these claims and extend the confidence of the approach, we

asked three students to conduct case studies on the IS approach to assess the approach's ability to detect vulnerabilities while remaining flexible to refinement. The study demonstrated the promise for future replication. The representative cases provided evidence that the approach was effective in architecture-centric testing for security. Nevertheless, the generalisation of the observations is subject to future work and extensive empirical studies.

The evaluation has explored the fitness of the approach in addressing the detection of hidden design vulnerabilities in the architecture model. In the first case, we took a web application's architecture and examined its security posture to ensure that it did not exhibit any unknown behaviours. We showed how the IS approach could be used to assess the security implications of any detected behaviour vulnerabilities. The importance of this case study is not in the architecture itself, but in how we used the concept of 'implied scenarios' as a way to detect hidden and malicious behaviours in the architecture. We verified the claim that the IS approach can affect the security posture of the system, and needs to be tested and addressed through refinement. In the second case study, we showed how the detection of ISs could provide insight into ways of assisting the architects to make informed refinements to secure the architecture. We also verified that the IS approach is adaptive, and thus is capable of responding to functional requirement changes and assessing how these changes can impact the security of the architecture. This case study involved the demonstration of how the IS approach could be used to test for security as the architecture evolves in an agile-driven development. We have also verified that the IS approach could be used to guide the selection process of candidate architectures with respect to the number of vulnerable behaviours each contains. We showed that the more hidden behaviours are identified, the more likely the architecture is to be abused, causing behaviours outside of the secure range.

The overall results show that the IS approach is capable of detecting security design vulnerabilities that can be exploited by attackers. We also studied its ability to systematically guide the architecture refinement process to address the threats detected.

We advanced our understanding of the security of architectures when addressed with an evolution-driven approach. We saw how the IS approach helped in understanding how refinements can break the security of the system, and how we can use the IS approach to assess the security of these changes. This issue of maintaining security while introducing change is thought to be challenging due to the ad-hoc nature of the way in which changes are handled [182]. The results also demonstrated that the approach was incremental and iterative, and the addition of new functionalities and the changing of the architecture was well-adapted. The applications of SecArch have drawn some preliminary observations regarding the approach's ability to be merged with existing approaches. We also learned that the search space of an IS can be enhanced to improve its ability to detect design vulnerabilities. Even though we looked at how the IS approach can be used to guide the selection of a secure architecture, research regarding which architecture is more secure than another is still required.

7

Discussion

In this Chapter, we will provide an evaluation of the approach described in this thesis and from the case studies in Section 6.3. Using the results and feedback gathered from the case study, and through our experience of conducting the studies, we will evaluate the steps and their results. The evaluation of the approach is split between theory and practical usage.

7.1 On Architecture-Centric Security Testing

The first aspect we discuss is the way in which the process we present in this thesis facilitates the security testing process at the architecture level. By architecture-centric security testing, we refer to the detection of emergent behaviours in the architecture model so that we can test whether the architecture behaves as intended. The testing process consists of the detection of malicious behaviours, which are scenarios that have security implications in regards to the architecture and the refinement of the architecture to guard against the occurrence of the behaviours. The goal is to reach a secure architecture that behaves as intended. The analogue between regular testing and architecture testing is the same; rather than working at the code level, debugging an application then fixing the code, we work on the architecture, debugging it by searching for malicious behaviours, then fixing it through refinement.

In Chapter 4, we discussed that the presence of ISs indicates limitations in the view of components about the desired global behaviour, and we saw that detecting ISs manually is a challenging task due to the computation demand it requires. Such demand is characterised by the concurrent nature of the software, where we need to realise the global behaviour of the system and all individual behaviours of the components/subsystems, as well as their composition together, in order to visualise the dynamic behaviour of the communication. In Chapter 3, we discussed the importance of understanding and checking the compositional security status of the architecture, and we highlighted that attackers intentionally probe unexpected interactions to make the system behave abnormally [177],

and that these unexpected interactions are design vulnerabilities [144] that need to be tested for. We then summarised that security testing aims to test for behaviours that are not explicitly specified. We used the concept of ISs as a means to the achievement of this goal.

7.1.1 Architecture-Centric

We saw that the IS approach, as architecture-centric testing, enabled us to abstract away unnecessary details where the focus is on the core functionality. We have demonstrated how security testing can begin at early stages of development, and that this is essential in order to detect design vulnerabilities (ie, design defects that result from incorrect design choices). The IS approach focused on testing the architecture design for abnormal and diverted behaviour. This design-specific approach allowed us to investigate the security of the design, and provided a systematic guide for testers to address the detected threats. Our experience with the case studies was that some of the refinements we made successfully addressed the threats, while other refinements introduced additional threats. Thus, we were able to keep track of the security status of the architecture and make informed decisions regarding the architecture. We saw through case studies that working at the architecture level helped in the minimisation of the impact of detected threats because we detected threats early, and refined the architecture before the threats manifested into the implementation. We used the threats detected to: (1) drive the refinement process to help us make informed decisions regarding the refinement while avoiding the breakage of the security of the architecture, as done in Case Study 1 and Case Study 3; and (2) guide the architecture selection process based on the level of the security threat detected in each architecture, as done in Case Study 3 (and Case Study 4 in Appendix B). We demonstrated that the IS approach provided a guided and incremental process for refinement without the need for architecture refactoring.

7.1.2 Composition

Throughout the applications we viewed in this Chapter, we studied various complications that arose from the functional composition of the specified behaviours, and we discussed the security impact of their presence on the architecture. Compositional security requires that components collaborate to ensure that global security is achieved. The IS approach took into account that not all compositions will be perfectly secure, and that searching for unexpected interactions between different components can reveal design vulnerabilities. The IS approach took the compositional level one step further as it allowed us to achieve adaptivity and incremental development as functionalities were composed. The outcomes of the case studies proved that ISs provided valuable information in the understanding of unforeseen behaviours, and in the detection of vulnerable compositions.

7.1.3 Proactivity

We used the IS approach as a means to proactively anticipate threats and take affirmative steps to deal with them positively by means of architecture refinement. We saw through case studies from different domains how starting the testing process at the architecture level provided early feedback in the prediction of the dynamic compositions of concurrent applications. The IS approach allowed us to take the lead in finding design vulnerabilities and addressing the threats before they manifested within the system. It allowed us to carry refinement by means of redesign to build security into the architecture. The additional advantage we gained in the application of the IS approach is that we did not rely on known threats; and since attackers use creative methods to attack the system, predetermined test cases are not effective in the detection of design vulnerabilities. Instead, we used the detection of emergent behaviour as the method for discovering threats based on the actual architecture design. This advantage of the IS approach is a significant advancement in security testing, where prediction is specific to the weaknesses in the architecture design. We were able to consider threats that were outside the range of normal use, and we

observed how the IS approach can find threats that we did not foresee. We saw in the refinements of Sections 6.3.1.4 and 6.3.3.6 that the IS approach allowed us to predict the security consequences of proposed refinements before they were committed.

7.2 On the Iteration

The principle of our approach is to provide a way to perform security testing at the architecture level in an iterative manner. This reflects a more realistic development of applications in which systems undergo many refinements before the system is built. In addition, this allowed us to test the security of the refinements before they were committed, as it is expected that changes will break the security of the existing architecture. We also took into account that, at the architecture level, information is often not provided as a whole upfront, especially in early stages of development. Reasons for that are varied, including the fact that designers and stakeholders might still be at the level of drawing out the requirements. Stakeholders may change their minds about the requirements, or there may be contradictions in the requirements. The level of changes in the requirements we have studied were twofold:

1. We studied (in Section 6.3.2) agile development to see how well our iterative and incremental approach could fit the purpose. This was done through the performance of two iterative cycles using different sets of requirements, where we modelled the addition of new requirements that requires integration with the existing iteration. The process begins by collecting related requirements that are to be implemented, then prioritising the requirements and then composing them incrementally (as done in the first agile iterative developed by the Industrial-Party) to produce the first working increment. The second iteration involved the addition of new requirements as the system evolved to produce the second working increment of the software. As we demonstrated, the testing process involved testing the increments both individually and when integrated with each other. The initial results presented very

promising insights into the approach's adaptivity to changes. Further testing may need to be conducted in this regard to enhance understanding and confidence in the approach.

2. We studied the refinement of the architecture through the addition and removal of existing functionalities in Case Studies 1 and 2.

The approach adapted to the changes and continued to evolve as propositioned. We feel that the approach has clear boundaries in terms of inputs required and outputs gained, which assists in the repetition. These inputs and outputs were described in detail in Chapter 4. For example, given the set of ISs detected from stage 1, these sets undergo analysis (stage 2) and architecture refinement (stage 3) as they are addressed. If stage 3 results in new ISs, then the next iteration will start once all ISs are addressed. We saw in the case studies in Sections 6.3.1.4 and 6.3.3.6 that continuous refinement as the design is built is more efficient than a single stage of refinement, which is processed at the end of each design phase. Thus, this allowed us to ensure that the architecture is refined with security in mind, rather than being evaluated for its security after the choices have been made and adopted, as well as to confirm with the requirements to check if the behaviour detected is required or not.

7.3 On the Applicability

A notable, desirable feature of the IS approach is its flexibility in application. It does not restrict the user to one security taxonomy for analysis, for example, nor does it prevent future extensions from being integrated into it. We have seen in Case Study 2 that the approach is flexible in incorporating additional techniques (ie, race condition analysis) into the approach to enhance the search space. We have specifically chosen to work on the architecture of the system because it offers an adequate level of generality. We have seen the approach's application in web applications [6], as presented in Section 6.3.1, on

the cloud [7], as presented in Section 6.3.2, and on a distributed smart camera system, as presented in Section 6.3.3. The applicability was discussed in the relevant Sections of the above case studies.

To further extend the confidence of the applicability claim, Appendix B documents two case studies on an identity management system [155]. In this application, the students used different approaches to classify the threats. We have seen that because many systems can be modelled in terms of communicating components, we were able to study various applications successfully. Our approach is suitable for architectures of systems deployed in unpredictable environments such as the cloud, where changes are unpredictable, and for distributed systems, where dynamic concurrent communication can be unpredictable. These systems share a common multi-threaded infrastructure that is subject to emergent behaviours –and which shares types of synchronous and asynchronous communications– in addition to the varying levels of synchronisation between components. Our approach is not suitable for centralised systems where one component centralises the communication with other components because these types of applications are highly synchronised. Moreover, most of these applications can be modelled in scenario-based specifications, which have been widely used in the industry for their flexibility and ease of use. Thus, we do not require additional training for testers. The students who performed some of the replica case studies did not need training to design the scenarios.

We use the architecture artefact because it is flexible enough to be used at different development stages (eg, maintenance) once the architecture is known. We saw that through the use of Case Studies 1 [6] and 3, we were able to extract scenarios from the code and construct the architecture from the scenarios to perform the case study. This indicates that the approach is flexible enough to test existing architectures and to construct architectures directly from requirements. For Case Study 2 [7] , we studied the two situations, where the first cycle of the iteration had the code present and the second iteration had only the requirements present. The identity management system in Appendix 4 also relied on the requirements to construct the architecture [155]. This illustrates the flexibility of

Table 7.1: Comparison between the number of messages used to model the case studies, and their number of files.

Case Study	No. of Messages	No of classes	Lines of code
1	23	9 class files	1533
2 (for first cycle only)	51	36 class files	2739
3	114	21 class files	7695

the approach to be used as either the code or the specification to extract the scenarios for forming the architecture. However, because the testing phase is usually time-constrained, our approach is designed to promote early application of and testing for security. This supports early planning, which allows more time to be devoted to improving the quality of the security testing process in the testing phase.

7.4 On the Scalability

The scalability benefit cannot be overlooked. Table 7.1 shows the number of messages used in the testing of each case study against the number of classes we would have tested if we had used an implementation-level approach. We abstracted method bodies by using method signatures to represent the behaviours, and we also abstracted internal behaviours to model observable behaviours between components. Our use of architecture in the IS approach allowed us to omit unnecessary details and focus on the desired behaviours to ensure that the design achieves its functional requirements. We believe this is a cost-effective method for the prediction of runtime behaviour, without the need to perform live testing after the system is built.

On the LTSA-MSC tool level, we did not experience scalability issues in small-scale applications, such as the web application case study in Section 6.3.1 (and the Identity Management Case study B). However, in the cloud and distributed smart camera case study, the LTSA-MSC tool experienced a state-explosion problem when several scenarios were synthesised together. To address this problem, we broke down the hMSC model into smaller subsets of scenarios that have been broken down into smaller sets for analysis.

These sets require comprehensive combination to ensure that no IS is overlooked when a certain combination is omitted. This was found to be a time-demanding process due to the repetition of scenarios in different sets. Fortunately, although we have focused on using LTSA-MSC and UBET, our approach does not depend on their usage. In fact, any tool that is capable of composing scenarios and searching for hidden ISs can be adapted. These tools were used as a proof of concept to verify the theory of using an incremental approach for security testing at the architecture level. However, this problem has the benefit of supporting the breakdown of the system into groups of requirements, either to perform detailed dynamic analysis for certain groups of functionalities (eg, areas of particular vulnerability), or to evolve the requirements for a specific functionality or subsystem. Once a group of functionalities is determined, our approach supports the composition of these functionalities incrementally to reach a larger subsystem.

In regards to the scalability of SecArch, we discovered that in practice, because we are relying on architecture traces, the trace-set can grow exponentially with respect to the size of the input model [107]. We have seen in Case Study 2 (Figure 6.9) that 72 traces needed to be inspected, which is a very time-consuming manual process. This applied equally to the repetitive detection of some race conditions across similar scenario compositions. However, both issues can be significantly addressed through automation. We will need further case studies to inspect how much reduction will be made with regards to the number of trace testers using automation.

7.5 Effectiveness

The proposed IS approach is geared towards detecting threats at the architecture level that arise due to a functional composition behaviour. We saw in the case studies that the complexity of integrating a functionality can lead to an unforeseen behaviour that affects the overall security of the system. Using the IS approach, we have tested a large number of possible interactions between collaborating components to ensure that the

overall desired behaviour is secure. Thus, the testing of each component independently is not sufficient for finding IS threats, which is where the IS approach plays an essential role in detecting composition-related threats. In this Section, we will review the effectiveness of the approach in the detection and refinement levels:

- At the detection level, throughout the application of the IS approach, we found that it detected critical situations with respect to the application domain. We reviewed the ISs detected for all the studies in Section 6.3 in detail. We found 6 ISs out of 16 total ISs in Case Study 1; 21 ISs out of 54 total ISs in Case Study 2; and 13 ISs out of 46 total ISs in Case Study 3. If we consider Case Study 3, even though the number of false positives is larger than the detected threats, the types of threats we found were significantly more important for the smart camera domain. Threats ranged from the performance of malicious exhaustive computations designed to waste a camera's energy, the malicious claiming ownership of objects, the sending of false information to other cameras and the interception of the network. The impact of such threats reveals the importance of investigating compositional security, because it relates directly to the design of the functionality. We saw in the case studies that the IS approach can be proactive in finding design vulnerabilities without requiring the use of vulnerability databases or similar tools that rely on past knowledge. We have previously discussed that learning from the past only leaves us behind attackers, whereas what is needed, considering the rapid development of attacks, is the development of approaches that are naturally proactive in their search for threats. The advantages gained at the compositional and at the design level may justify the effort required, depending on the criticality of the system under test.

- At the refinement level, we saw in Sections 6.3.1.4 and 6.3.3.6 how IS drove the refinement process to help us make informed decisions about the refinement, while taking into account that we must avoid breaking the security of the architecture. We demonstrated that each detected IS was used to guide the refinement process to ensure that the IS did not occur. We subjected each refinement cycle to further

testing to ensure that: (1) the IS is no longer possible; and (2) that the refinement does not result in new ISs. We have seen in Case Study 1 in Table 6.4 that the refinements were carried out to address each of the ISs detected, and we experienced a new Security IS for one of the refinements and made another refinement to address it. For Case Study 3 in Section 6.3.3.6, we witnessed a drop in the number of positive ISs, which indicates that the number of gaps in the specification had also dropped. This provides assurance in the refinements made to achieve a better set of specifications for the system. However, we experienced three Security ISs in response to the refinements. For example, we found that we needed another functionality to perform negotiations between two cameras that were tracking the same object. We also found that our refined mechanism for broadcasting messages to all cameras, when a camera owns an object, introduced another problem whereby an object may be lost and detected by a camera that does not track it because it believes that the object is still tracked by the camera that sent the announcement. These examples demonstrate how the systematic refinement brought insight into the security of the decisions made regarding the refinement. This test-driven and incremental refinement allowed us to perform informed refinements without the need for major architecture refactoring.

However, evaluating approaches similar to the IS approach is rather difficult, as their effectiveness is dependent on the way in which practitioners apply them. In particular, security can be subjective to the application and highly dependent on the environment [156]. The type of application determines the types of IS threats detected because the threats detectable using the IS approach are design-specific; for example, the threats detected in a cloud setting, where the environment is unknown, are more complicated than in the testing of closed systems where environment variables are known, as in the first case study on web applications. The change of domain also reflected significantly on the types of threats detected. We found that in Case Study 1 (web application) and Case Study 2 (cloud application), where we dealt with data, corruption of information was the

leading threat; whereas in Case Study 3 (distributed smart camera system), the leading threat was denial of service. As a result, the effectiveness of its application is subject to the context in which the IS approach is applied. The IS approach is also open; it can be easily integrated to complement existing security testing approaches with the objective of detecting design-level vulnerabilities, while the existing approaches (cited in Chapter 2) can be used to detect different levels of vulnerabilities. This integration is likely to provide a comprehensive prediction of the security of the under-test architecture.

7.6 Threat to Validity

Humans provide more subtle checks and fewer false positives and negatives [2] as compared to automated tools. The IS approach relies on a balanced use of human involvement and automation. We use automation to detect emergent behaviours so that the humans involved are not driven by their experience when searching for threats. In addition, the detection of ISs requires high-level computations that make human detection of ISs a challenging, error-prone process. On the other hand, security experts cannot be replaced by automated tools [45], because attackers work in artistically different manners [148]. The security experts use creative and novel approaches to find emergent behaviours in the system [62], and so security flaws cannot be easily generalised. Thus, to counteract such creativity we require human involvement that can defend against the attackers' creativity through creative means. Some attackers try to learn about the system's assets by understanding how the system functions [148]. This behaviour is often launched by insiders who happen to have more access to knowledge and information than typical outsiders. They often aim to trigger unexpected paths in the system [116], which is precisely what the IS detection approach aimed to achieve in order to ease the process for testers. The IS approach depicts the search-type approach for attacks, for which we find vulnerabilities and then seek to understand their impact. In the applications of the IS approach, we presented the testers with a set of emergent behaviours that can take place

in the under-test architectures, and used that information to project how an attacker could abuse the system through that weakness. We then used that information to guard against the occurrence of weaknesses. That way, we provided testers with the necessary information required to protect the system from attackers. The interpretation of results is thus enhanced by using the concept of the ISs as opposed to relying solely on the experience of the testers.

Tools used in the application process may be inaccurate. We have taken care to select established tools that have been publically accessible and used for long period of time. These tools have been used in a number of publications (Eg, LTSA-MSC was used in [187, 147, 59], UBET was used in [150, 22, 24]).

7.7 Limitations on the Modelling-Level

Our present approach is limited in its use of MSC specifications due to limitations in the LTSA-MSC tool. We have seen in Sections 6.3.1.4 and 6.3.3.6 that we were not able to model the internal behaviour of components. We have mentioned previously that IS detection is not concerned with internal behaviour, and that it deals with how components interact with each other (ie, external communication). However, this hinders the modelling of the refinement process, which means we will need to find other means to document the changes so that testers do not overlook the refinement. This also hinders the race condition detection, because the internal behaviour will not be taken into consideration when checking for race conditions. On the modelling level, we were also unable to represent data dependencies between components, which therefore constrains interaction descriptions as a type of message rather than as a value of the message being passed between partners. In compositional design this is not highly detrimental to the conciseness of interactions, because we were able to change the message name to represent its difference in content. These limitations are not inherent to our approach, because the LTSA-MSC tool can be replaced with any other tool that is capable of detecting ISs.

Another limitation is that the set of MSCs used to model the system can become complex to manage, especially with the detection of ISs. This complexity is exhibited when a high number of alternative scenarios evolve for describing and sequencing. For example, if a concurrent set of five interactions is permissible in a section of a composition, then the designer must describe each alternative case for invocation and reply to each. This was particularly apparent in the distributed smart camera case study in Section 6.3.3, in which we had a number of valid responses returned by each camera.

7.8 Discussion of Tool Support

As discussed in Section 6.2 of the thesis, the approach makes a novel use of a number of sound tools to assist the process of IS detection, classification and architecture refinement for architecture-level testing for security. Nevertheless, these tools were not previously exploited to the benefit of Security IS detection and architecture refinement. The following subsection evaluates the extent to which these tools can support the automation of our approach and we reflect on its strengths and limitation. We appropriately split the discussion into a series of criteria for evaluation taken from the work in [36]. These criteria consider support from several view-points including: ease of learning, early payback, efficiency of developers time, increase in benefits, error detection, integrated development environment enabled, focus on analysis and support for evolutionary development.

7.8.1 Ease of Learning

'Notations and tools should provide a starting point for writing formal specifications for developers who would not otherwise write them. The knowledge of formal specifications needed to start realizing benefits should be minimal' [36].

Our approach aims at providing the following criteria for its ease of learning and carrying out the security testing process:

Stage 1: IS detection

- The design specifications are based on the scenario elicitation approach, and the use of bMSC and hMSC sequence charts is widely used and understood in industry. A graphical interface is used so that the user does not have to learn new notations.

- The LTSA-MSC tool performs the detection of IS, and the detected ISs are returned in the MSC specification. Users of the tool do not need to learn how ISs are detected and synthesised. We have observed that from a brief explanation (approximately 30 minutes) regarding the usage of the LTSA-MSC tool, the students who conducted the case study in Appendix B were independently able to move into the security evaluation and refinement.

- By identifying vulnerabilities through the detection of ISs, we are able to explicitly link the IS to the requirements and assets that introduce them into the system. Testers can *visualise* the vulnerable components of the system through the studying of the ISs detected, examine how its occurrence impacts the system, trace security breaches back to the source vulnerability and relate vulnerabilities to the attackers that can launch the threat. The IS approach has the advantage of visualising the locations of vulnerabilities to aid in the: (1) understanding the emergence of vulnerabilities; (2) understanding how threats can be compromised; and (3) allowing for the application of countermeasures during the design [69].

Stage 2: Classifying IS This process is simplified in two ways: (1) The testers have a scenario model of the emergent behaviour, and their role is to answer: *If the IS behaviour executes, what security implication will it cause on the system assets? Can the IS be misused to cause a security breach?* Comparatively, this is easier than asking testers to search for threats manually, and then attempt to understand how compositions can cause insecure behaviours. We have simplified this further by: (2) using an intuitive classification taxonomy [70] that allows testers to address this phase systematically and

attach a malicious story behind the scenario. This taxonomy allows testers to think of various ways in which an attack may be launched, as it provides insight into possible combinations of threat *attackers*, in addition to possible *outcomes* of a successful attack.

SecArch Enhancement The use of SecArch is optional, and is provided to enhance the search-space of the IS approach. First, the SecArch requires architecture traces to be extracted, and the LTSA-MSC tool has a built in animated label transition system interface to extract these traces. The second part requires testing for race conditions, and we use a graphical tool called UBET that takes in MSC charts and reports back on the detected race conditions graphically. These steps can be fully automated to speed up the process. Malicious race conditions undergo the same process as that of Stage 2.

Stage 3: Architecture Refinement The refinement process is drawn from an understanding of how the threats occur. With the available classification of the IS, the testers need to answer: *How do we prevent the occurrence of the IS?* We saw in the case studies in Section 6.3 that this process is systematic and is guided by the allowance of testers to study the refinements, then to test the security of the refinement by re-assessing the architecture. This provided insights into: (1) how secure that change had been, and whether it had been able to address previously detected ISs; and (2) whether that new change had broken a previously corrected problem.

7.8.2 Early Payback

'Methods and tools should provide significant benefits almost as soon as people begin to use them' [36].

Early payback is a key objective of the approach. As we discussed in Chapters 1 and 3, in terms of motivation for this work, our aim is to support the detection of malicious behaviours during the early stages of development so that testers can address design vulnerabilities before the system is built. We chose to work at the architecture-centric

level because early feedback provides assurance that the changes that took place in the architecture (the addition new functionalities or the removal of them) can be verified. An example of this is demonstrated in Case Study 1 in Section 6.3.1.4, in which every refinement considered was assessed for the potential emergence of ISs, and when ISs occurred, they were either: (1) positive and required no further refinement; or (2) negative Security ISs that required further refinements until no further emergent behaviours arose. This allowed us to receive instant security feedback on the changes in the architecture.

We also experienced early feedback when testers were able to study with the presence of negative scenarios in the system, to validate the architecture's resilience to attacks. We have seen in Case Study 2 in Section 6.3.2.4 that the replay of detected race conditions into the architecture allowed us to detect the further emergence of ISs. This early feedback supports testers in those situations in which refinement cannot take place, possibly due to restrictions on time and budgets. As a result, testers can continue to investigate how that unwanted behaviour could impact the system, and possibly find other measures to protect the system (such as implementation level patches, if applicable).

7.8.3 Efficiency

'Tools should make efficient use of a developer's time. Turnaround time with an interactive tool should be comparable to that of normal compilation. Developers are likely to be more patient, however, with completely automatic tools that perform more extensive analysis' [36].

There are currently three points of interests in regards to efficiency:

- *On the efficiency related to the speed of feedback.* (We discussed this in Section 7.8.2).
- *On the engineering part of the approach.* On the engineering part of the approach. We have not given an in-depth analysis on the efficiency of our approach in this regard because, as was discussed in Section 6.3.2.4, there are a number of processes

that can be automated to enhance efficiency and overcome their current limitations. These were not implemented due to timing constraints of the PhD research. Examples of these are: (1) The translation of scenarios back and forth, from the LTSA-MSC (XML) used to detect ISs, into UBET (MS common console document). Currently, we have created all race condition traces manually. (2) The re-assessments of previously assessed race conditions in other scenarios. Our experience with the approach showed that each stage is simple in its requirements and executions, and that the manual stages (2 and 3) require human involvement because the process requires accurate interpretations and architecture refinement. Further refinements/improvements are expected to be carried in order to automate the manual processes stated above. Clearly, our approach relies heavily on and is limited by the efficiency of the underlying model-checking technology.

- *On the false positives.* In our case studies, we experienced a number of false positives. We used these positive ISs to confirm the accuracy of the required behaviour and that the overall behaviour satisfied the functional requirements. In Case Study 1 in Section 6.2.2, we detected 10 false positives; in Case Study 2 in Section 6.2.2, we detected 31; and in Case Study 3 in Section 6.2.3, we detected 33. These false positives relied primarily on the level of synchronisation between components. In Case Study 1, we had a higher level of synchronisation between the communicating components; whereas in the second case study the environment of the cloud was unpredictable, and thus the synchronisation level between the components was low. The third case study represented the distributed smart cameras, and the synchronisation level between components was also low. Synchronisation allows the components to be aware of their position in the execution trace. When components are aware of each other's movements, an IS can be prevented because the components will not divert their behaviour. However, high synchronisation may not be desirable due to energy requirements, as we have seen in the distributed smart camera case study in Section 6.2.3.

7.8.4 Integrated Use

'Methods and tools should work in conjunction with each other and with common programming languages and techniques. Developers should not have to buy into a new methodology completely to begin receiving benefits. The use of tools for formal methods should be integrated with that of tools for traditional software development. Eg, compilers and simulators' [36].

When we designed our approach, we took into account the following: (1) the approach should be easily integrated with other approaches. We have seen in Case Study 2 that we were able to integrate our approach with Alur's algorithm to enhance our approach's search space. This provided insight into the extendibility of the approach. In Appendix B, the students chose to use their own taxonomies to conduct the IS classification. This provided an insight into the approach's ability to integrate with the existing classifications of organisations. We also saw –for example, in Case Study 1– that the refinement process was flexible and allowed a prioritisation process to proceed before the refinement stage began, so that the order of refinements could change depending on the importance of the refinement; and (2) we have chosen to work with the scenario-based specification language because it is widely used in the industry to promote communication between developers and the stakeholders, and can thus be integrated with many of the existing evaluation techniques that utilize scenario-based specifications.

Our approach is not intended to replace existing security testing techniques that focus on testing the security of an information system after the system is implemented. We were unable to detect or consider attacks related to specific implementations at the time of the design. On the contrary, our process aims to complement such implementation-related testing techniques and provide a guided approach that enables developers to: (1) identify important security vulnerabilities related to the design models of the system at an early stage in the developmental process; and (2) provide insight into the refinement of the system in order to overcome identified security vulnerabilities, and to ensure that the design of the system enforces the necessary security requirements.

On the technical level, the LTSA-MSC tool can be used on different IDE development frameworks, such as Eclipse and Netbeans IDEs. Thus, these various editors are supported and can be used as long as they conform with the MSC specifications.

7.8.5 Incremental Gain for Incremental Effort

'Benefits should increase as developers get more adept or put more effort into writing specifications or using tools' [36].

The general incremental and elaboration support of the approach allows incremental benefits of the approach. The most apparent example of this is the SecArch, in which we have gained significant benefits to the incorporation of race conditions to search for further ISs. We discovered that we were able to detect 14 more ISs in addition to the 7 we detected without incorporating race conditions. The results supported by the other two case studies show that: (1) the more input we provide to the approach, the more likely we are to detect more ISs; and (2) the process can lead to no IS detection if the refinement process effectively addresses the causes of IS; consequently, the benefit is measured by the assurance it provides to the testers in regards to the behaviour of the architecture; and finally (3) we will gain a better understanding of the system's dynamic behaviour.

7.8.6 Evolutionary Development

'Methods and tools should support evolutionary system development by allowing partial specification and analysis of selected aspects of a system' [36].

Due to the nature of IS compositions, they may represent only one part of the architecture. As we discussed in Section 7.2, it is clear that our approach provides an incremental, elaborative approach to building compositions and realising the effects of changes on security as they are introduced. We saw in the case studies that we were able to assess the entire architecture when the tool scaled, as in Case Study 1, and we saw when it was necessary to break down the system into smaller sub-Sections in order to

evaluate the parts individually, as was done in Case Study 2. In Case Study 2, we saw the way in which we broke the system into iteration 1 and iteration 2, as is done in agile developments to support partial specification and analysis. This gives an insight into the approach's ability to assess systems as a whole, and systems that are made of smaller sub-systems, as was done in Case Study 3.

7.8.7 Orientation Towards Error Detection

'Methods and tools should be optimised for finding errors, not for certifying correctness. They should support generating counterexamples as a means of debugging' [36].

The approach is geared towards vulnerability detection. ISs are emergent behaviours that are possible in the dynamic view of concurrent systems, but are not perceived from a static view of the specifications. Thus, the essence of the approach is the highlighting of inconsistencies between the desired global behaviour and the local behaviour of components. Such inconsistencies cause faults and vulnerabilities to arise during the composition of local component views. Our use of IS supports the concept of 'debugging' as counterexamples in the detection of ISs, and in using them to systematically refine the architecture.

7.8.8 Multiple Use

'It should be possible to amortize the cost of a method or tool over many uses. For example, it should be possible to derive benefits from a single specification at several points in a program's life cycle: in design analysis, code optimisation, test case generation, and regression testing' [36]

Our use of IS focuses on the design vulnerability detection caused by unperceived diversions of system behaviour. We showed in the case studies that the IS approach is proactive in finding design vulnerabilities for three situations: (1) the discovery of ISs during the general security testing of the architecture; (2) the assessment that the

refinement process does not introduce new design vulnerabilities; and (3) the selection of the most secure architecture based on the number of design vulnerabilities detected in each. Even though we have not tried to construct test cases that result from the IS traces, approaches exist that can construct test cases automatically from scenarios, such as [9, 152].

7.9 Summary

Our discussion in this chapter explored the satisfaction of the requirements set in Chapter3 and how the IS approach ensured that each requirement is satisfied. We looked at the threat to validity, its potential to scale, limitations in the modelling, and we took the discussion further to the approach's ease of use, its potential for multiple usage, and its efficiency. In the next Chapter, we wrap up the thesis and highlight future work.

8

Conclusion and Future Work

In this chapter, we reflect on the contribution of this thesis, and highlight future directions for the use of IS approach.

8.1 Summary of Contribution

This thesis presents a novel architecture-centric testing approach for security, which uses *implied scenarios* (ISs) [185] to detect design vulnerabilities in the software architecture. It explores the field of security testing to probe for greater understanding of the state-of-the-art practices in the testing of software systems for security. Further, it reviews the approaches that practitioners and researchers have considered in security testing. The review has established the evidence that the field of security software testing has lacked systematic guidance, and has only been performed on low-level software artefacts such as design and code. The approaches have been found to be limited in their scalability and adaptivity to changes in functional requirements. Their use of predefined specifications and test cases renders the meaningfulness and effectiveness of security testing 'myopic'. This is because: (i) the landscape of security tends to evolve with changes in the threat landscape; (ii) many emerging behaviours, which can threaten the security posture of the systems, occur at runtime and are therefore difficult to anticipate at design time; (iii) while code and low-level design artefacts tend to be effective for testing programming bugs and assembling components into security subsystems, many of the vulnerabilities are architectural in nature [76, 114].

The thesis explores the requirements for architecture-centric security testing. It introduces a novel solution for the detection of hidden behaviours within the architecture. Moreover, the thesis introduces the concept of security ISs as unanticipated (possibly malicious) behaviours that indicate potential insecurities in the architecture. It utilises the architecture as the appropriate level of abstraction in order to tackle the complexity of testing and to provide the potential for scalability, which in turn allows for the testing of large-scale, complex applications. Further, this use of architecture supports the critical

consideration of secure composition, interaction, concurrency and emerging behaviour. A three-phased method for security testing is proposed: (1) as the detection of design-level vulnerabilities in the architecture in an incremental manner, it is done via investigating emergent behaviours (ie, ISs) in the composition of evolving functionalities; (2) by classifying the impact of detected ISs on the security of the architecture; (3) utilising the detected ISs and their impacts to guide the refinement of the architecture. The approach takes into account that refinements might introduce new emergent behaviours. Consequently, the refinement process is test-driven and incremental, such that refinements are tested before they are committed. This method provides proactive and early feedback on the security of any changes applied to the architecture, which allows testers to make informed decisions about the refinements.

The thesis also presents SecArch, an extension to the IS approach, to enhance its search space to detect hidden race conditions. SecArch is focused on the prediction of further valid conditions, in the face of real parallelism in distributed systems, with respect to non-FIFO queues. The enhancement proposes moving from purely dynamic LTS behaviour models to structural MSC models in order to preserve the structural properties that are used to detect race conditions. These race conditions are used to investigate the impact of the presence of malicious behaviours within the application. This can be measured by observing the number of additional ISs that arise as a result of the addition of malicious race conditions to the architecture.

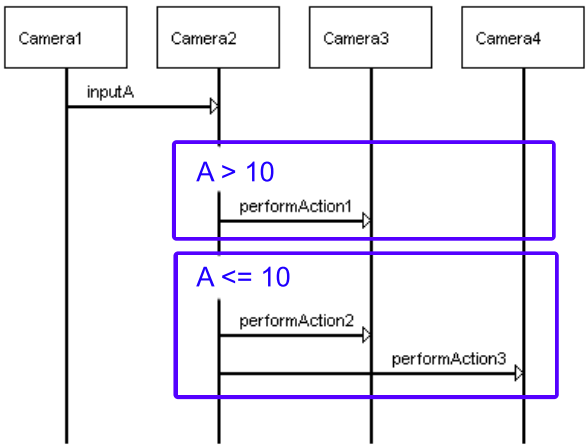
This thesis reports on the applications of the proposed approach and its extension to three case studies for testing the security of distributed and cloud architectures in the presence of uncertainty in the operating environment, the unpredictability of interactions and possible security ISs. The applications demonstrate novelty in the way in which security testing is used to address emergent behaviour in applications characterised by dynamism, heterogeneity, openness, scale and unpredictability in their operational and evolutionary trends. We have drawn on these case studies to evaluate the thesis. We explored the fitness of the approach for detecting threats in the architecture and guiding the

architecture refinement process. We experimented with the approach's ability to inform the selection of a more secure architecture, and we demonstrated that security testing can be performed even when we do not have complete knowledge about the behaviour of the components. We verified the claim that ISs are behaviours that can result in security implications for the architecture, and that testing for these ISs is critical in order to predict the compositional security of the dynamic behaviour of the architecture. We demonstrated that the detection ISs supports a proactive application of security testing, in which we detect design vulnerabilities based on the architecture itself, without relying on pre-known threats already explored by attackers. We have verified that security testing can be performed in iterative development cycles; this allowed us to build security into the design by making informed decisions about the impact of changes made to the architecture. We have also reflected on the potential of the approach with respect to criteria like ease of learning, effectiveness, scalability and applicability to further evaluate the thesis.

8.2 Future Work

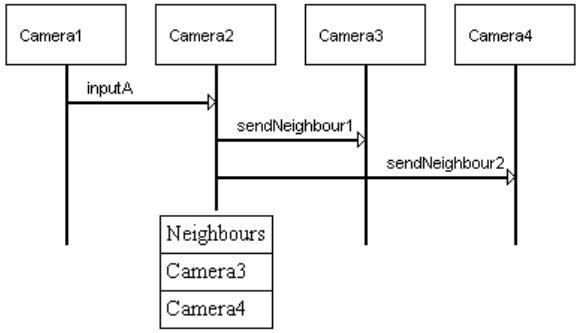
1. Our current approach has considered ISs that are caused by control flow rather than data flow. A component may decide its action based on the value of some input. For example, it is necessary to take into account: (a) single input, single output structure; (b) single input, multiple output structure; and (c) multiple input, multiple output structure. Each of these structures may have different implications on the ISs detected. Consider the example presented in Figure 8.1. The figure illustrates two alternative conditions in which a component might receive an input of some value and, based on the value, decide which action to take. The illustration models an incident featuring single input with single output, and single input with multiple outputs. Using current semantics, we cannot model alternative conditions based on the input value.

Figure 8.1 Demonstrating example of alternative scenarios based on the value of some input.



2. In the case studies, we found that we were not able to test the internal data structure of cameras automatically. This information is essential in order to predict the correct behaviours of the cameras and the internal choices available to them. We found that understanding the impact of internal data structures was needed to reflect realistic system behaviours. Consider the example presented in Figure 8.2. The figure illustrates four communicating cameras where Camera 1 sends a message to Camera 2, and Camera 2 forwards the message to its neighbours. The modelling of this scenario required that we know the identity of the neighbouring cameras. Consequently: (1) when we detected ISs, we had to ensure that the IS did not violate the neighbouring relationship (ie, a camera does not communicate with another camera outside its list of neighbours); and (2) every scenario had to be modelled individually to reflect the neighbouring relationships, which can be a cumbersome and error-prone process. It would be desirable to create one abstract scenario, from which the rest could be generated with respect to different possible neighbouring relationships, and search for subsequent ISs. Furthermore, we mentioned in the distributed camera case study that the cameras learn about their neighbouring cameras as the network evolves. The ability to check internal data structures supports the performance of ISs in parallel with the learning process of neighbouring cameras.

Figure 8.2 Demonstrating example of internal structure of cameras to enhance the modelling scope.



3. Further research can be done with respect to the dynamics of moving objects (ie, change in the state of objects). This could imply that an unpredictable movement of objects might cause components to act outside of their normal uses. The IS approach might be enhanced through the observation of the various ways in which an object could move, as well as the detection of ISs when a component acts abnormally.

4. Currently, we are dealing with a fixed set of components, which may or may not interact depending on the scenario. One very interesting application of the IS approach within the cloud would be the analysis of the impact of the dynamic composition of different services into a cloud-based architecture. Dynamic composition entails changing components in the architecture (adding/removing) at the runtime. It is necessary to analyse how such changes might impact the system; for example, by detecting an emergent behaviour that arose due to some dynamic change. This opens up opportunities for making decisions regarding which service should be selected with the least number of emergent behaviours. An example of this may relate to the privacy-aware composition of cloud services, for which we may be able to better decide which of the services will provide the level of privacy we are seeking (eg, service providers such as Google and Yahoo may advertise services for a login service or certificate-authentication-based component). Because all of these services have different privacy agreements, the selection of one will determine the overall privacy we can achieve. An IS could be used to inform the selection/composition decision

so that if security ISs are detected, then the choice should be geared towards a more secure component.

5. We can use probabilistic measures to state the likelihood of an IS trace's execution. This can be used in a number of situations:

   - We have seen that the number of positive ISs can be large. To address this, we can use probabilistic measures to determine the probability of the IS execution, and use that probability to prioritise the ISs. That way, the likely ISs can be examined first.
   - If, for example, we know that a component has an 80% security level, then we can say that using this component's composition will increase the chance of an attack occurring. This would allow us to choose which component is less likely to result in the execution of an attack.
   - In the selection of the architecture, the probability of execution can tell us a lot about the security of the architecture. For example, an architecture may have fewer threats, but a higher likelihood of attack execution. From a business perspective, this architecture might be worse than an architecture with more threats that are highly unlikely to execute.
   - It will also give us an indication as to whether the refinement is worth pursuing. If the threat is unlikely to execute, it probably is not worth the refinement.

6. We observed the ability of the IS approach to highlight potential trade-offs between non-functional requirements (eg, between security and usability as in Case Study 1, and between security and utility as in Case Study 3). We examined the way in which the IS approach can guide the refinement process with respect to security, and we acknowledged that we had to compromise between ensuring security and usability/utility. We are working on strengthening our understanding in this field by performing multi-objective trade-off analyses of non-functional requirements, in order to investigate the ability of ISs to inform the trade-off process.

7. The concept of an IS can be applied in game theory. In a game, we might have the option to decide which move to pursue next, and the ability to question whether a particular move would allow us to gain more information beneficial for winning the game. ISs could be used to explore the search space so that we can determine the outcome of making a particular move. If we will detect ISs, then we are able to learn more about the game state. For example, this can be used to determine whether there is a more beneficial move available and whether that move could be used to increase some utility in the game. We have used the concept of ISs with respect to negative behaviours, but in the case of game theory it can be used in its positive meaning to reflect new, desirable information. Thus, the presence of ISs can indicate possible cheats/hints available in the game.

8. Possible future work could focus on the automatic refinement of the model. That is, when a behaviour diverts, we know that the local component is missing some information about the global communication (ie, it may need to be synchronised with other components). It might be possible to devise an algorithm that synchronises components automatically to prevent this diversion from occurring and provide a suggested refinement to the testers, who could either accept or reject it.

8.3 Closing Remarks

This thesis makes a novel and timely contribution to the field of security software engineering by presenting the first architecture-centric testing approach for security –in the absence of closely related work. This thesis demonstrates that architecture-centric testing provides a proactive means for the detection of design vulnerabilities by guiding the refinement of architectures when building security into their designs. The application of the approach on various case studies demonstrates novelty, timeliness and potential in the way testing is addressed in emerging domains (eg, cloud, distributed cameras), which are characterized by dynamism, heterogeneity, openness, scale and unpredictability in their

operational and evolutionary trends. This contribution has the following implications: (i) advancement of the understanding of the potential of using the architecture as an artefact for performing security testing due to the benefits described in the earlier chapters; and (ii) stimulation and motivation of future research in the area of architecture-centric testing for security, threat detection and application of security ISs in emerging domains.

Bibliography

[1] A. Abdurazik and J. Offutt. Using uml collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard*, UML'00, pages 383–395, Berlin, Heidelberg, 2000. Springer-Verlag.

[2] M. Abi-Antoun and J. M. Barnes. Analyzing security architectures. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 3–12, New York, NY, USA, 2010. ACM.

[3] C. R. Aguayo González and J. H. Reed. Power fingerprinting in sdr integrity assessment for security and regulatory compliance. *Analog Integrated Circuits and Signal Processing*, 69(2-3):307–327, December 2011.

[4] G.-J. Ahn and M. E. Shin. Role-based authorization constraints specification using object constraint language. In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '01, pages 157–162, Washington, DC, USA, 2001. IEEE Computer Society.

[5] S. Al-Azzani and R. Bahsoon. Semi-automated detection of architectural threats for security testing. In *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ESEC/FSE Doctoral Symposium '09, pages 25–26, New York, NY, USA, 2009. ACM.

[6] S. Al-Azzani and R. Bahsoon. Using implied scenarios in security testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 15–21, New York, NY, USA, 2010. ACM.

[7] S. Al-Azzani and R. Bahsoon. Secarch: Architecture-level evaluation and testing for security. In *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA-ECSA '12, pages 51–60, Washington, DC, USA, 2012. IEEE Computer Society.

[8] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. Oxford University Press, 1977.

[9] I. Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 20(1):58–66, January 2003.

[10] I. F. Alexander. Initial industrial experience of misuse cases in trade-off analysis. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 61–70, Washington, DC, USA, 2002. IEEE Computer Society.

[11] S. Ali, L. C. Briand, M. J.-u. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem. A state-based approach to integration testing based on uml models. *Information and Software Technology*, 49:1087–1106, November 2007.

[12] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, and N. R. Mead. *Software Security Engineering: A Guide for Project Managers (The SEI Series in Software Engineering)*. Addison-Wesley Professional, 1 edition, 2008.

[13] W. Allen, C. Dou, and G. Marin. A model-based approach to the security testing of network protocol implementations. In *Proceedings 2006 31st IEEE Conference on Local Computer Networks*, pages 1008–1015, Tampa, FL, 2006. IEEE.

[14] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 304–313, New York, NY, USA, 2000. ACM.

[15] R. Alur, G. J. Holzmann, and D. Peled. An analyser for mesage sequence charts. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAs '96, pages 35–48, London, UK, UK, 1996. Springer-Verlag.

[16] O. E. Ariss, J. Wu, and D. Xu. Towards an enhanced design level security: Integrating attack trees with statecharts. In *Proceedings of the 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[17] N. Associates. Cybercop scanner. `www.iss.net/security_center/reference/vuln/CyberCop_Scanner.htm`.

[18] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 260–275, London, UK, UK, 2003. Springer-Verlag.

[19] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[20] C. W. Axelrod. The need for functional security testing. *The Journal of Defense Software Engineering*, 24(2):17–21, March 2011.

[21] R. Bahsoon and W. Emmerich. Evaluating architectural stability with real options theory. In *Proceedings of the 2004 IEEE International Conference on Software Maintenance*, pages 443–447. IEEE Computer Society, 2004.

[22] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell. Automatic generation of conformance tests from message sequence charts. In *Proceedings of the 3rd International Conference on Telecommunications and Beyond: The Broader Applicability of SDL and MSC*, SAM'02, pages 170–198, Berlin, Heidelberg, 2003. Springer-Verlag.

[23] J. Bergey. *A Proactive Means for Incorporating a Software Architecture Evaluation in a DoD System Acquisition.* Technical note. Carnegie Mellon University, Software Engineering Institute, 2009.

[24] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:85–97, January 2005.

[25] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE Transactions on Computers*, 52(4):492–505, April 2003.

[26] M. Blackburn, R. Busser, A. Nauman, and R. Chandramouli. Model-based approach to security test automation. In *Proceeding of Quality Week*. The National Institute of Standards and Technology, 2001.

[27] M. Borrett. Web application security qa. `ftp://ftp.software.ibm.com/software/ae/events/doc/rational_appscan.pdf`, 2007.

[28] J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, and F. Schadle. Model-based testing of cryptographic components – lessons learned from experience. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 192–201, Washington, DC, USA, 2013. IEEE Computer Society.

[29] S. Bracher and P. Krishnan. Enabling security testing from specification to code. In *Proceedings of the 5th International Conference on Integrated Formal Methods*, IFM'05, pages 150–166, Berlin, Heidelberg, 2005. Springer-Verlag.

[30] L. Briand, Y. Labiche, and Y. Liu. Combining uml sequence and state machine diagrams for data-flow based integration testing. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Strrle, and D. Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 74–89. Springer Berlin Heidelberg, 2012.

[31] L. C. Briand and Y. Labiche. A uml-based approach to system testing. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 194–208, London, UK, UK, 2001. Springer-Verlag.

[32] CENZIC. Application security trends report q1 2008. `https://info.cenzic.com/rs/cenzic/images/Cenzic-white-paper-Cenzic-Vulnerability-Report.pdf`, 2012.

[33] S. A. Christopher Allen, Kalle and D. Holmberg. Edit this page. `http://sourceforge.net/projects/editthispagephp/`.

[34] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, USA, 1989. IEEE Press.

[35] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[36] R. Cleaveland and S. A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, 1996.

[37] F. Dadeau, P.-C. Héam, and R. Kheddam. Mutation-based test generation from security protocols in hlpsl. In *Proceedings of the 2011 Fourth IEEE International*

Conference on Software Testing, Verification and Validation, ICST '11, pages 240–248, Washington, DC, USA, 2011. IEEE Computer Society.

[38] H. Dai, C. Murphy, and G. Kaiser. Confu: Configuration fuzzing testing framework for software vulnerability detection. *International Journal of Secure Software Engineering*, 1(3):41–55, July 2010.

[39] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[40] M. H. Diallo, J. Romero-mariona, S. E. Sim, T. A. Alspaugh, and D. J. Richardson. A comparative evaluation of three approaches to specifying security requirements. In *Proceedings of the Twelfth Working Conference on Requirements Engineering: Foundation for Software Quality*. Elsevier, 2006.

[41] M. S. Dias and M. E. R. Vieira. Software architecture analysis based on statechart semantics. In *Proceedings of the 10th International Workshop on Software Specification and Design*, IWSSD '00, pages 133–, Washington, DC, USA, 2000. IEEE Computer Society.

[42] T. Dimitrakos, B. Ritchie, D. Raptis, J. O. Aagedal, F. d. Braber, K. Stølen, and S. H. Houmb. Integrating model-based security risk management into ebusiness systems development: The coras approach. In *Proceedings of the IFIP Conference on Towards The Knowledge Society: E-Commerce, E-Business, E-Government*, I3E '02, pages 159–175, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[43] A. Drappa and J. Ludewig. Simulation in software engineering training. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 199–208, New York, NY, USA, 2000. ACM.

[44] P. Duffy. Exploratory testing and analysis of full ceramic ball bearings. *SAE Transactions*, 33(5):267, 1991.

[45] L. Dukes, X. Yuan, and F. Akowuah. A case study on web application security testing with tools and manual testing. In *Proceedings of the IEEE Southeastcon*, pages 1–6, Jacksonville, FL, 2013.

[46] B. Dunphy. Key considerations for outsourcing security. `http://trygstad.rice.iit.edu:8000/Articles/KeyConsiderationsForOutsourcingSecurity-HNS.pdf`, May 2004.

[47] M. Ekstedt and T. Sommestad. Enterprise architecture models for cyber security analysis. In *Proceedings of the 2009 IEEE Power Systems Conference and Exposition*, pages 1 –6, Seattle, WA, march 2009. IEEE Power and Energy Society.

[48] G. Elahi and E. Yu. A goal oriented approach for modeling and analyzing security trade-offs. In *Proceedings of the 26th international conference on Conceptual modeling*, ER'07, pages 375–390, Berlin, Heidelberg, 2007. Springer-Verlag.

[49] E. Elkind, B. Genest, and D. Peled. Detecting races in ensembles of message sequence charts. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 420–434, Berlin, Heidelberg, 2007. Springer-Verlag.

[50] M. Essafi, L. Labed, and H. B. Ghezala. S2d-prom: A strategy oriented process model for secure software development. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA '07, pages 24–30, Cap Esterel, 2007. IEEE Computer Society.

[51] L. Esterle, P. Lewis, M. Bogdanski, B. Rinner, and X. Yao. A socio-economic approach to online vision graph generation and handover in distributed smart camera networks. In *Proceedings of the 5th ACM/IEEE International Conference on Distributed Smart Cameras*, pages 1–6, Ghent, 2011. IEEE.

[52] D. T. et al. Symantec internet security threat report. `http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_internet_security_threat_report_xi_03_2007.en-us.pdf`, Volume XI, Published March 2007.

[53] F. Faniyi, R. Bahsoon, A. Evans, and R. Kazman. Evaluating security properties of architectures in unpredictable environments: A case for cloud. *Working IEEE/IFIP Conference on Software Architecture*, 1:127–136, 2011.

[54] F. Farahmand, S. B. Navathe, G. P. Sharp, and P. H. Enslow. A management perspective on risk of security threats to information systems. *Information Technology and Management*, 6(2-3):203–225, April 2005.

[55] M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle*, pages 109–114. Xpert Publishing Services, 2011.

[56] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997.

[57] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 90–96, New York, NY, USA, 2001. ACM.

[58] A. Flo and A. Josang. Consequences of botnets spreading to mobile devices. In *Proceedings of the 14th Nordic Conference on Secure IT Systems*, pages 37–43, Berlin, Heidelberg, 2009. Springer-Verlag.

[59] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ws-engineer: A model-based approach to engineering web service compositions and choreography. In L. Baresi and E. Nitto, editors, *Test and Analysis of Web Services*, pages 87–119. Springer Berlin Heidelberg, 2007.

[60] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the 2007 IEEE Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[61] J. Friedman. *Dictionary of Business and Economic Terms*. Barron's Educational Series, 5th edition, 2012.

[62] V. Gandotra, A. Archana Singhal, and P. Bedi. Layered security architecture for threat management using multi-agent system. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–11, September 2011.

[63] V. Gandotra, A. Singhal, and P. Bedi. Identifying security requirements hybrid technique. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 407–412, Washington, DC, USA, 2009. IEEE Computer Society.

[64] B. Gates. Gates highlights progress on security, outlines next steps for continued innovation. `http://www.microsoft.com/en-us/news/press/2005/feb05/02-15rsa05keynotepr.aspx`, 2005.

[65] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[66] O. M. Group. Unified modeling language (uml), version 2.2. `http://www.omg.org/technology/documents/formal/uml.htm`.

[67] N. Guelfi and A. Saidane. Seter: Towards architecture-model based security engineering. *International Journal of Secure Software Engineering*, 3(3):23–49, July 2012.

[68] R. Gula. Broadening the scope of penetration-testing techniques. `www.forum-intrusion.com/archive/ENTRASYS.pdf`, 1999.

[69] M. A. Hadavi, H. Shirazi, H. M. Sangchi, and V. S. Hamishagi. Software security; a vulnerability activity revisit. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, ARES '08, pages 866–872, Washington, DC, USA, 2008. IEEE Computer Society.

[70] S. Hansman and R. Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43, 2005.

[71] K. He, Z. Feng, and X. Li. An attack scenario based approach for software security testing at design stage. In *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology*, volume 1 of *ISCSCT '08*, pages 782–787, Washington, DC, USA, 2008. IEEE Computer Society.

[72] M. J. Heavner, D. R. Fatland, E. Hood, and C. Connor. Seamonster: A demonstration sensor web operating in virtual globes. *Comput. Geosci.*, 37(1):93–99, January 2011.

[73] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack. Threat modeling: Uncover security design flaws using the stride approach. *The Microsoft Journal for Developers MSDN Magazine*, 1:6, November 2006.

[74] P. Herzog. Open source security testing methodology manual (osstmm). `http://www.isecom.org/mirror/OSSTMM.3.pdf`, December 2010.

[75] T. Heyman, K. Yskout, R. Scandariato, and W. Joosen. An analysis of the security patterns landscape. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 3–10, Washington, DC, USA, 2007. IEEE Computer Society.

[76] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley Software Security Series, 2004.

[77] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive can networks - practical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, 96(1):11–25, 2011. ¡ce:title¿Special Issue on Safecomp 2008¡/ce:title¿.

[78] J. Howard and T. Longstaff. *A Common Language for Computer Security Incidents*. Sandia National Laboratories, 1998.

[79] M. Howard. A look inside the security development lifecycle at microsoft. `http://msdn.microsoft.com/en-us/magazine/cc163705.aspx`, 2005.

[80] M. Howard and D. C. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, 2002.

[81] D. Hughes, P. Greenwood, and G. Coulson. A framework for testing distributed systems. In *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, P2P '04, pages 262–263, Washington, DC, USA, 2004. IEEE Computer Society.

[82] S. Institute. Critical controls for effective cyber defense. `http://www.sans.org/critical-security-controls/cag4.pdf`.

[83] J. Itkonen and K. Rautiainen. Exploratory testing: a multiple case study. In *Proceedings of the 2005 IEEE International Symposium on Empirical Software Engineering*, pages 84–93, Noosa Heads, Australia, November 2005. IEEE.

[84] ITU. Message sequence charts. Technical report, International Telecommunications Union. Telecommunication Standardisation Sector., 1996.

[85] M. G. Jaatun and I. A. Tøndel. Covering your assets in software engineering. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, ARES '08, pages 1172–1179, Washington, DC, USA, 2008. IEEE Computer Society.

[86] L. Jinhua and L. Jing. Model checking security vulnerabilities in software design. In *6th International Conference on Wireless Communications Networking and Mobile Computing*, pages 1–4, 2010.

[87] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *Proceedings of the 3rd international workshop on Automation of software test*, AST '08, pages 41–44, New York, NY, USA, 2008. ACM.

[88] L. Jurani. Using fuzzing to detect security vulnerabilities. Technical report, INFIGO, 2006.

[89] J. Jürjens. Umlsec: Extending uml for secure systems development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, UK, 2002. Springer-Verlag.

[90] J. Jürjens. Model-based security testing using umlsec. *Electron. Notes Theor. Comput. Sci.*, 220(1):93–104, December 2008.

[91] H. M. K. Goertzel, T. Winograd and P. Holley. B. hamilton, security in the software lifecycle. Technical report, Department of Homeland Security, August, Version 1.2 2006.

[92] R. Kalman. On the general theory of control systems. *IRE Transactions on Automatic Control*, 4(3):110–110, 1959.

[93] P. Karpati, A. L. Opdahl, and G. Sindre. Experimental comparison of misuse case maps with misuse cases and system architecture diagrams for eliciting security vulnerabilities and mitigations. In *Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security*, ARES '11, pages 507–514, Washington, DC, USA, 2011. IEEE Computer Society.

[94] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[95] e. a. Kent Beck, Mike Beedle. Manifesto for agile software development. `http://agilemanifesto.org/`, 2001.

[96] e. a. Kent Beck, Mike Beedle. Principles behind the agile manifesto. `http://agilemanifesto.org/principles.html`, 2001.

[97] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. *IEEE Proceedings Software Engineering*, 146(4):187–192, August 1999.

[98] R. Kissel. Glossary of key information security terms. `http://csrc.nist.gov/publications/nistir/ir7298-rev1/nistir-7298-revision1.pdf`, February 2011.

[99] J. Kong and D. Xu. A uml-based framework for design and analysis of dependable software. In *Proceedings of the 2008 32Nd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '08, pages 28–31, Washington, DC, USA, 2008. IEEE Computer Society.

[100] H. kuk Kim, Y. seo Choi, and D. il Seo. Design of attack generation test-suite based scenario for security solutions testing. In *Proceedings of the 7th International Conference on Advanced Communication Technology*, volume 1, pages 676–679, Phoenix Park, 2005.

[101] D. Kundu and D. Samanta. A novel approach to generate test cases from uml activity diagrams. *Journal Of Object Technologies*, 8(3):65–83, May 2009.

[102] B. Labs. Ubet. `http://cm.bell-labs.com/cm/cs/what/ubet/`.

[103] M.-A. Laverdiere, A. Mourad, A. Hanna, and M. Debbabi. Security design patterns: Survey and evaluation. In *Proceedings of the 2009 IEEE Canadian Conference on Electrical and Computer Engineering*, pages 1605–1608, Ottawa, Ont, 2006. IEEE Computer Society.

[104] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte. Model-based vulnerability testing for web applications. In *Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 445–452, Luxembourg, March 2013.

[105] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Monitoring and control in scenario-based requirements analysis. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 382–391, New York, NY, USA, 2005. ACM.

[106] X. Li and K. He. A unified threat model for assessing threat in web applications. In *Proceedings of the 2008 International Conference on Information Security and Assurance*, ISA '08, pages 142–145, Washington, DC, USA, 2008. IEEE Computer Society.

[107] B. Lindstrom, P. Pettersson, and J. Offutt. Generating trace-sets for model-based testing. In *Proceedings of the 18th IEEE International Symposium on Software Reliability.*, pages 171–180, Nov 2007.

[108] T. Lodderstedt, D. A. Basin, and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441, London, UK, UK, 2002. Springer-Verlag.

[109] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258, 2013.

[110] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, volume 0, pages 161–166, Los Alamitos, CA, USA, 1987. IEEE Computer Society.

[111] G. McGraw. *Software Security: Building Security In...* Addison-Wesley Professional, 2006.

[112] G. McGraw. Software security. *Datenschutz und Datensicherheit - DuD*, 36(9):662–665, 2012.

[113] G. McGraw and B. Potter. Software security testing. *IEEE Security and Privacy*, 2(5):81–85, September 2004.

[114] G. McGraw and J. Viega. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley Professional, 2001.

[115] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, SP '94, pages 79–85, Washington, DC, USA, 1994. IEEE Computer Society.

[116] P. McMinn. Search-based software test data generation: A survey: Research articles. *Software Testing, Verification & Reliability*, 14(2):105–156, June 2004.

[117] P. H. Meland and J. Jensen. Secure software design in practice. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, ARES '08, pages 1164–1171, Washington, DC, USA, 2008. IEEE Computer Society.

[118] W. Michael, C. & Radosevich. Risk-based and functional security testing. Technical report, Build Security In., 2005.

[119] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[120] B. Mitchell. Inherent causal orderings of partial order scenarios. In *Proceedings of the First International Conference on Theoretical Aspects of Computing*, ICTAC'04, pages 113–127, Berlin, Heidelberg, 2005. Springer-Verlag.

[121] B. Mitchell. Resolving race conditions in asynchronous partial order scenarios. *IEEE Transactions on Software Engineering*, 31(9):767–784, 2005.

[122] G. Mogyorodi. Requirements-based testing: An overview. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, TOOLS '01, pages 286–295, Washington, DC, USA, 2001. IEEE Computer Society.

[123] A. Morais, E. Martins, A. Cavalli, and W. Jimenez. Security protocol testing using attack trees. In *Proceedings of the 2009 International Conference on Computational Science and Engineering*, volume 2 of *CSE '09*, pages 690–697, Washington, DC, USA, 2009. IEEE Computer Society.

[124] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 537–552, Berlin, Heidelberg, 2008. Springer-Verlag.

[125] H. Mouratidis and P. Giorgini. Security attack testing (sat)-testing the security of information systems at design time. *Information Systems*, 32:1166 –1183, 2007.

[126] H. Mouratidis, M. Weiss, and P. Giorgini. Modeling secure systems using an agent-oriented approach and security patterns. *International Journal of Software Engineering and Knowledge Engineering*, 16(3):471, 2006.

[127] M. Mozaffari-Kermani and A. Reyhani-Masoleh. Concurrent structure-independent fault detection schemes for the advanced encryption standard. *IEEE Transactions on Computers*, 59(5):608–622, May 2010.

[128] H. Muccini. Detecting implied scenarios analyzing non-local branching choices. In *Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering*, FASE'03, pages 372–386, Berlin, Heidelberg, 2003. Springer-Verlag.

[129] P. V. Murthy, P. C. Anitha, M. Mahesh, and R. Subramanyan. Test ready uml statechart models. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, SCESM '06, pages 75–82, New York, NY, USA, 2006. ACM.

[130] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.

[131] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3(2):58–62, May 2005.

[132] N. I. of Standards and T. (NIST). Cve and cce statistics query page. `http://web.nvd.nist.gov/view/vuln/statistics`.

[133] E. A. Oladimeji, S. Supakkul, and L. Chung. Security threat modeling and analysis: A goal-oriented approach. In *Proceedings of the 10 th IASTED International Conference on Software Engineering and Applications*. Acta Press Inc,# 80, 4500-16 Avenue N. W, Calgary, AB, T 3 B 0 M 6, Canada,, 2006.

[134] A. L. Opdahl and G. Sindre. Experimental comparison of attack trees and misuse cases for security threat identification. *Information and Software Technology*, 51(5):916–932, May 2009.

[135] V. Paradigm. Visual paradigm for uml. `http://www.visual-paradigm.com/aboutus/award/`.

[136] P. A. Pari Salas, P. Krishnan, and K. J. Ross. Model-based security vulnerability testing. In *Proceedings of the 2007 Australian Software Engineering Conference*, ASWEC '07, pages 284–296, Washington, DC, USA, 2007. IEEE Computer Society.

[137] J. J. Pauli and D. Xu. Misuse case-based design and analysis of secure software architecture. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, volume 1-2 of *ITCC '05*, pages 398–403, Washington, DC, USA, 2005. IEEE Computer Society.

[138] K. P. Peralta, A. M. Orozco, and A. F. Z. F. M. Oliveira. Specifying security aspects in uml models, 2008.

[139] C. Phillips and L. P. Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*, NSPW '98, pages 71–79, New York, NY, USA, 1998. ACM.

[140] M. Popovic and I. Velikic. A generic model-based test case generator. In *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, ECBS '05, pages 221–228, Washington, DC, USA, 2005. IEEE Computer Society.

[141] W. Prenninger and A. Pretschner. Abstractions for model-based testing. *Electron. Notes Theor. Comput. Sci.*, 116:59–71, Jan. 2005.

[142] A. Pretschner, T. Mouelhi, and Y. L. Traon. Model-based tests for access control policies. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 338–347, Washington, DC, USA, 2008. IEEE Computer Society.

[143] M. A. Qadeer, A. Iqbal, M. Zahid, and M. R. Siddiqui. Network traffic analysis and intrusion detection using packet sniffer. In *Proceedings of the 2010 Second International Conference on Communication Software and Networks*, ICCSN '10, pages 313–317, Washington, DC, USA, 2010. IEEE Computer Society.

[144] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1-2):189–209, July 2002.

[145] I. Ray, N. Li, R. France, and D.-K. Kim. Using uml to visualize role-based access control constraints. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 115–124, New York, NY, USA, 2004. ACM.

[146] S. Rehman and K. Mustafa. Research on software design level security vulnerabilities. *ACM SIGSOFT Software Engineering Notes*, 34(6):1–5, December 2009.

[147] G. Rodrigues, D. Rosenblum, and S. Uchitel. Using scenarios to predict the reliability of concurrent component-based software systems. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 111–126. Springer Berlin Heidelberg, 2005.

[148] S. Rohrig and S. S. Ag. Using process models to analyze health care security requirements, 2002.

[149] D. Rosenzweig, D. Runje, and W. Schulte. Model-based testing of cryptographic protocols. In R. Nicola and D. Sangiorgi, editors, *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 33–60. Springer Berlin Heidelberg, 2005.

[150] A. Roychoudhury, A. Goel, and B. Sengupta. Symbolic message sequence charts. *ACM Transactions on Software Engineering and Methodology*, 21(2):12:1–12:44, March 2012.

[151] I. Rus, F. Shull, and P. Donzelli. Decision support for using software inspections. In *Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop*, pages 3–11, Greenbelt, Maryland, December 2003. IEEE Computer Society.

[152] J. RYSER and M. GLINZ. Scent: A method employing scenarios to systematically derive testcases for system test. Technical report, University of Zurich, 2000.

[153] M. Sahinoglu. Security meter: A practical decision-tree model to quantify risk. *IEEE Security and Privacy*, 3(3):18–24, May 2005.

[154] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[155] A. N. Sarah Al-Azzani and R. Bahsoon. Architecture-centric testing for security: An agile perspective. In M. A. Babar, A. W. Brown, K. Koskimies, and I. Mistrik, editors, *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Elsevier, 1st edition, 2013.

[156] L. A. C. Sascha Konrad, Betty H.C. Cheng and R. Wassermann. Using security patterns to model and analyze security requirements. In *Proceedings in the 2003 IEEE Workshop on Requirements for High Assurance Systems*, pages 13–22. IEEE Computer Society, 2003.

[157] B. Schneier. Attack trees - modeling security threats. *Dr. Dobbs journal of Software Tools*, 24(12):21–29, 1999.

[158] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM.

[159] N. Shahmehri. The shield project. `http://www.shields-project.eu/?q=node/19`.

[160] H. Shahriar and M. Zulkernine. Music: Mutation-based sql injection vulnerability checking. In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[161] H. Shahriar and M. Zulkernine. Mutec: Mutation-based testing of cross site scripting. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09, pages 47–53, Washington, DC, USA, 2009. IEEE Computer Society.

[162] M. E. Shin and H. Gomaa. Software requirements and architecture modeling for evolving non-secure applications into secure applications. *Science of Computer Programming*, 66(1):60 – 70, 2007. Special Issue on the 5th International Workshop on System/Software Architectures (IWSSA'06).

[163] D. Shreyas. Software engineering for security: Towards architecting secure software. *Information and Computer Science Department*, 1:1–7, 2002.

[164] G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *Requir. Eng.*, 10(1):34–44, January 2005.

[165] B. Skaggs, B. Blackburn, G. Manes, and S. Shenoi. Network vulnerability analysis. In *Proceedings of the 45th Midwest Symposium on Circuits and Systems*, volume 3, pages 493–495, 2002.

[166] A. Sodiya, S. Onashoga, and B. Oladunjoye. Threat modeling using fuzzy logic paradigm. In *Issues in Informing Science & Information Technology*, volume 4, pages 53–60. Informing Science Institute, 2007.

[167] F. C. d. Sousa, N. C. Mendonca, S. Uchitel, and J. Kramer. Detecting implied scenarios from execution traces. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 50–59, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[168] C. Staite. Mobile advertising: Privacy and reward. `http://www.cs.bham.ac.uk/~cxs548/papers/auction.pdf`, 2011.

[169] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[170] S. Stolfo, S. Bellovin, and D. Evans. Measuring security. *IEEE Security Privacy*, 9(3):60–65, 2011.

[171] A. Sutcliffe. Scenario-based requirements engineering. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering*, RE '03, pages 320–327, Washington, DC, USA, 2003. IEEE Computer Society.

[172] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, Redmond, WA, USA, 2004.

[173] I. S. Systems. Internet scanner (iss). Web, 2003.

[174] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[175] G. Taleck. Ambiguity resolution via passive os fingerprinting. In G. Vigna, C. Kruegel, and E. Jonsson, editors, *Recent Advances in Intrusion Detection*, volume 2820 of *Lecture Notes in Computer Science*, pages 192–206. Springer Berlin Heidelberg, 2003.

[176] A. K. Talukder, V. K. Maurya, B. G. Santhosh, J. Ebenezer, S. V. Muni, K. P. Jevitha, S. Samanta, and A. R. Pais. Security-aware software development life cycle (sasdlc): processes and tools. In *Proceedings of the Sixth international conference on Wireless and Optical Communications Networks*, WOCN'09, pages 253–257, Piscataway, NJ, USA, 2009. IEEE Press.

[177] H. H. Thompson. Why security testing is hard. *IEEE Security and Privacy*, 1(4):83–86, July 2003.

[178] H. H. Thompson, J. A. Whittaker, and F. E. Mottay. Software security vulnerability testing in hostile environments. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 260–264, New York, NY, USA, 2002. ACM.

[179] I. Tndel, J. Jensen, and L. Rstad. Combining misuse cases with attack trees and security activity models. In *Proceedings of the 10th IEEE International Conference on Availability, Reliability, and Security*, pages 438–445. IEEE Computer Society, 2010.

[180] I. Tondel, M. Jaatun, and P. Meland. Security requirements for the rest of us: A survey. *IEEE Software*, 25(1):20–27, 2008.

[181] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 93–102, Washington, DC, USA, 2007. IEEE Computer Society.

[182] T. T. Tun, Y. Yu, C. B. Haley, and B. Nuseibeh. Model-based argument analysis for evolving security requirements. In *Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 88–97, 2010.

[183] S. Türpe. Security testing: Turning practice into theory. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 294–302, Washington, DC, USA, 2008. IEEE Computer Society.

[184] G. M. Uchenick and W. M. Vanfleet. Multiple independent levels of safety and security: high assurance architecture for msls/mls. In *Proceedings of the 2005 IEEE Military Communications Conference*, pages 610–614. IEEE, 2005.

[185] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 597–601, Berlin, Heidelberg, 2003. Springer-Verlag.

[186] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. *ACM SIGSOFT Software Engineering Notes*, 26(5):74–82, September 2001.

[187] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 109–118, New York, NY, USA, 2002. ACM.

[188] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, January 2004.

[189] M. Ventuneac. Apache struts 2 multiple reflected xss in xwork error pages. `http://seclists.org/bugtraq/2011/May/81`, May 2011.

[190] L. Wang, E. Wong, and D. Xu. A threat model driven approach for security testing. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[191] D. A. Wheeler. Applying engineering principles to system security design & implementation. In *Proceeding of The Annual Computer Security Applications Workshop for Application of Engineering Principles to System Security Design*, volume 1, page 6, Anaheim, CA, 2002. The Annual Computer Security Applications.

[192] J. Whittle, D. Wijesekera, and M. Hartong. Executable misuse cases for modeling security concerns. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 121–130, New York, NY, USA, 2008. ACM.

[193] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. In *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '02, pages 471–482, London, UK, UK, 2002. Springer-Verlag.

[194] J. M. Wing. A call to action: Look beyond the horizon. *IEEE Security and Privacy*, 1(6):62–67, November 2003.

[195] D. Xu and K. E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278, Apr. 2006.

[196] D. Xu and J. J. Pauli. Threat-driven design and analysis of secure software architectures. *Journal of Information Assurance and Security*, 1(3):171–180, 2006.

[197] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He, and Z. Li. A model-based fuzz framework to the security testing of tcg software stack implementations. In *Proceedings of the 2009 International Conference on Multimedia Information Networking and Security*, volume 1 of *MINES '09*, pages 149–152, Washington, DC, USA, 2009. IEEE Computer Society.

[198] J. Yeo. Using penetration testing to enhance your company's security. *Computer Fraud & Security*, 2013(4):17–20, 2013.

[199] R. K. Yin. *Case study research: Design and methods*, volume 5. Sage, 2009.

[200] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In *Proceedings of the 4th Pattern Languages of Program Design*, pages 301–336. Addison-Wesley, 2000.

[201] C. Zou. *Modeling, early detection, and mitigation of internet worm attacks.* PhD thesis, University of Massachusetts Amherst, 2005. AAI3193965.