Building High Assurance Secure Applications using Security Patterns for Capability-based Platforms

by

Paul Rimba

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

March, 2016

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Rimba

First name: Paul **Other name/s**: -

Abbreviation for degree as given in the University calendar: PhD

School: Computer Science and Engineering **Faculty**: Engineering

Title: Building High Assurance Secure Applications using Security Patterns for Capability-based Platforms

Abstract 350 words maximum: (PLEASE TYPE)

Building a secure software system is difficult and requires significant expertise and effort. A secure system requires a secure design, a secure implementation of that design, and a secure platform on which the implementation executes. Furthermore, it must also provide assurances about its security properties. Security patterns have been proposed to help the design of secure systems. However, security patterns are written independently of the specifics of the underlying platforms. This leaves a gap between security patterns and the underlying platform. Furthermore, composition of security patterns is challenging because each pattern uses different design elements and may target different security requirements.

The aim of this research is to improve our understanding of the design of high assurance secure applications. The main contributions of this thesis are a pattern-based composition approach to incrementally build and verify application designs. The approach reuses security knowledge from security patterns, and security mechanisms from secure underlying platforms. I propose the concept of a design fragment as an instantiation of a security pattern for a specific platform. This allows for design-level verification to provide assurance about security properties. Six primitive operations are provided for composition and are proven to preserve confidentiality. A collection of 277 security patterns from existing literature is synthesized. Each pattern is defined in a new security pattern template which is based on previous pattern templates.

The contributions are evaluated using two case studies from different domains, a Continuous Deployment (CD) pipeline and an electricity Smart Meter. These case studies show that the approach applies across different domains. The design fragments and their verification procedures are reusable and the composition tactics are sufficient to express steps in the design of a secure software system.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

Signature	Witness	Date
.....	.....	03/03/2016

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .....

Date 03/03/2016

COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.
I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).
I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed .....

Date 03/03/2016

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed .....

Date 03/03/2016

List of Publications by Thesis Author

The work presented in this thesis has resulted in the following formal publications. Thesis author, Paul Rimba, is the main contributor of the papers where he is the first author. He also presented the conference papers at the respective conference venues.

1. Paul Rimba. Building High Assurance Secure Applications Using Security Patterns for Capability-based Platforms. In *Proceedings of the 35th International Conference on Software Engineering*, 2013, San Francisco, California, USA. IEEE Press.

    - The thesis author is the sole author of this paper. The main idea in this paper was developed by the thesis author and was improved with the help of his supervisors, Dr. Liming Zhu and Dr. Mark Staples. The work in this paper is reported throughout the thesis.

2. Paul Rimba, Liming Zhu, Len Bass, Ihor Kuz and Steve Reeves. Composing Patterns to Construct Secure Systems. In *Proceedings of the 11th European Dependable Computing Conference*, 2015, Paris, France. IEEE Computer Society.

    - The thesis author is the main contributor to this paper. He wrote the majority of the paper and performed the modelling and analysis. The main idea was developed by the thesis author and was improved with suggestions and feedbacks from the co-authors. The co-authors also improved the quality of the

writing and formal notation in the paper. The work in this paper is reported throughout the thesis.

3. Len Bass, Ralph Holz, Paul Rimba, An Binh Tran and Liming Zhu. Securing a Deployment Pipeline. In *Proceedings of the 3rd International Workshop on Release Engineering*, 2015, Firenze, Italy. IEEE Computer Society.

    - The thesis author helped to write several sections of the paper and contributed to the idea in this paper, which is partly based on Rimba, Zhu, Bass, Kuz and Reeves (2015). The work in this paper is reported in Chapter 7.

4. Paul Rimba, Liming Zhu, Xiwei Xu and Daniel Sun. Building Secure Applications using Pattern-Based Design Fragments. In *Proceedings of the 34th International Symposium on Reliable Distributed Systems*, 2015, Montreal, Canada. IEEE Computer Society.

    - The thesis author wrote the majority of the paper and performed the modelling and analysis. The main idea was developed by the thesis author and his supervisors. The co-authors improved the quality of the writing in the paper. The work in this paper is reported in Chapter 7.

5. Paul Rimba, Liming Zhu, Mark Staples and Steve Reeves. Property Preserving Composition Tactics for Building Secure Systems. *to be submitted*.

    - The thesis author is the main contributor to this paper. He wrote the majority of the paper. The formal notation was improved with the help of the co-authors. The co-authors also helped to improve the quality of the paper.

Dedicated to God, my parents and my brother

Abstract

Building a secure software system is difficult and requires significant expertise and effort. A secure system requires a secure design, a secure implementation of that design, and a secure platform on which the implementation executes. Furthermore, it must also provide assurances about its security properties. Security patterns have been proposed to help the design of secure systems. However, security patterns are written independently of the specifics of the underlying platforms. This leaves a gap between security patterns and the underlying platform. Furthermore, composition of security patterns is challenging because each pattern uses different design elements and may target different security requirements.

The aim of this research is to improve our understanding of the design of high assurance secure applications. The main contributions of this thesis are a pattern-based composition approach to incrementally build and verify application designs. The approach reuses security knowledge from security patterns, and security mechanisms from secure underlying platforms. I propose the concept of a design fragment as an instantiation of a security pattern for a specific platform. This allows for design-level verification to provide assurance about security properties. Six primitive operations are provided for composition and are proven to preserve confidentiality. A collection of 279 security patterns from existing literature is synthesized. Each pattern is defined in a new security pattern template which is based on previous pattern templates.

The contributions are evaluated using two case studies from different domains, a Continuous Deployment (CD) pipeline and an electricity Smart Meter. These case studies show that the approach applies across different domains. The design fragments and their verification procedures are reusable and the composition tactics are sufficient to express steps in the design of a secure software system.

Acknowledgements

First of all, I would like to thank my Heavenly Father who taught me that in Him, all things are possible. He is my cornerstone and provider in times of darkness during the course of this thesis.

I would like to thank my parents for instilling principles of life and perseverance in me, which is essential for this journey. I owe my deepest gratitude to my parents for their enormous sacrifice to be apart from me in order to allow me pursue my future. Also, I would like to thank my brother, Johan, for his unconditional support and encouragement during the course of my PhD.

I am most indebted to my two supervisors Dr. Liming Zhu and Dr. Mark Staples for their conscientious and dedicated supervision over the years. Their guidance and wisdom were invaluable, which have improved this work tremendously more than I can say. I want to thank Dr. Zhu for trusting me to embark on this journey with complete freedom to explore different possibilities and for the support to attend different events and conferences. In addition, I would like to thank Dr. Staples for patiently examining my dissertation. I could not have asked for better supervisors.

I would also like to thank Prof. Len Bass for his guidance over the years in NICTA, especially in the initial stages in defining the research directions. I am thankful for Dr. Ihor Kuz for allowing me to be a part of the security architecture project, for the guidance and for examining parts of my thesis. I would also like to thank Dr. Toby

Murray for the valuable discussions about security and formalism and moral support. I want to also thank Prof. Steve Reeves for the opportunity to collaborate on the formalism of the composition tactics in Chapter 5.

I would like to express my gratitude to the anonymous reviewers of the papers published. They have provided invaluable suggestions and comments, which have vastly improved this research.

I would like to thank the co-authors of all my publications. Your invaluable suggestions, comments and perceptions have taught me patience, open-mindedness and professionalism.

A number of people have provided enormous support to me during the course of this thesis. My heartfelt appreciation goes to my friends, both in Sydney and back home, for their love and support. I am grateful for these wonderful folks: Anthony Kotanto, Adi Kurniawan, Daniel Dermawan, Michelle Widjaja, Yufi Tukiaty, and Stephanie Gunawan, whose constant support have kept me sane and pushed me to completion. My colleagues at NICTA, Dr. Xiwei Xu, Dr. Liang Zhao, Dr. Daniel Guimarans, Dr. Daniel Sun, Dongyao Wu, An Binh Tran, and Dr. Dana Kusumo, you have been a constant source of inspiration and encouragement. To my friends and colleagues whose names are not mentioned, you are also an essential part of this journey and thank you for adapting your schedule to accommodate mine and for bearing with me over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building a secure software system is difficult and requires significant expertise and effort. A secure system requires a secure design, a secure implementation of that design, and a secure platform on which the implementation executes. Furthermore, it is not sufficient that an system be secure, it must also be seen to be secure. That is, the argument that a system is secure is essential for providing assurance about the level of security of a system. I examine these four elements in slightly more detail now.

- *Secure platform* — A secure platform provides the foundation for building large security-critical systems (Klein et al., 2009; Neumann and Watson, 2010; Woodruff et al., 2014). Such a platform consists of a combination of hardware and software that provide security mechanisms and policies usable by the system. While this provides a solid base for building secure systems, it does not by itself guarantee that systems constructed on top of this platform are secure.

- *Secure design* — The use of security patterns (Yoder and Barcalow, 1997), tactics (Bass et al., 2012) and best practices can help in the design of security-critical software systems. A security pattern is an encapsulation of expert knowledge and best practices in the area of secure software design (Fernandez-Buglioni, 2013).

While security patterns can reduce required expertise, there are several difficulties in applying them. Firstly, a single security pattern normally cannot meet all the security requirements of an system. Therefore, a developer needs to apply multiple security patterns to a design. Composition of security patterns is challenging because each pattern may target different security requirements and have its own elements (e.g. actors involved and their interactions). Secondly, security patterns are written independently of the specifics of the underlying platforms. This leaves a gap between security patterns and the underlying platform.

- *Secure implementation* — Systems that have strong security requirements also require assurance that their implementation satisfy those requirements. Formal verification can provide a high level of assurance. However, formally verifying a large complex system is very costly. Therefore, reducing the effort of verification by verifying the system design at an early stage of system development can be beneficial (Kuz et al., 2012). Furthermore, verifying the design might result in system-level properties if there are good isolation guarantees provided by the platform.

- *Security assurance* — A security-critical system requires assurances about its security properties. Formal verification can provide a high level of assurance. An assurance case is a way to structure informal arguments to provide evidence of a system's assurance. The safety-critical system community uses assurance cases extensively to structure arguments demonstrating safety claims about systems (Bloomfield and Bishop, 2010; Kelly, 1999). Assurance cases have also been used to support security claims. Weinstock et al. (2013) state that a security assurance case presents arguments, supported by evidence, of claims that systems exhibit certain security properties. Security assurance cases can rely on arguments or evidence derived from the use of analytical techniques and tools. These techniques

and tools can differ from one portion of the assurance case to another.

1.1 Research Aim and Questions

The aim of this research is to improve our understanding of high assurance secure application design utilizing existing security knowledge from security patterns, and security mechansims from secure underlying platforms. The following research questions guide research in this thesis:

RQ-1: How can I specialize security patterns as reusable design fragments targeting particular platforms?

RQ-2: How can I verify that security properties hold of these design fragments?

RQ-3: How can many design fragments be applied to the design and verification of secure software systems?

1.2 Research Contributions

The main contributions of this thesis can be summarized as follows:

A New Security Patterns Catalog

- A new security pattern template is proposed that are based on five commonly used pattern templates. I unify these templates by adopting common fields and merging the overlapping fields.
- A collection of 279 security patterns is synthesized from existing literature. This, to the best of my knowledge, is the largest collection of security patterns reported.

- An analysis of the metadata of the patterns in the catalog is performed. This provides insights into the spread of security properties addressed by the patterns and the trend of pattern proposal.

Capability-specific Design Fragments

- I propose the concept of a design fragment as an instantiation of a security pattern for a specific platform. This allows for design-level verification to provide assurance about its properties.

- I provide guidance on how to derive design fragments from security patterns.

Composition of Design Fragments

- I propose a pattern-based composition approach to incrementally build and verify an application design using verified security patterns and composition primitives for design fragments.

- I provide a proof that the composition primitives preserve security (confidentiality). This provides a formal basis to build more complex composition tactics that also preserve security.

- I describe examples of higher-level tactics (combination of composition primitives) that preserve security.

Verification Procedure

- I describe verification procedures for design fragments.

- I describe an abstract general security property, *Protected By*.

The contributions are evaluated using two case studies, a Continuous Deployment (CD) pipeline and a Smart Meter. The CD pipeline case study evaluates the applicability of the approach and the expressiveness of the composition primitives to compose systems together to build a secure system. The pipeline case study shows that my approach can secure a system in the presence of threats, defined in a threat model. The case study also indicates that the composition primitives are sufficient to express design steps required to secure the pipeline. The Smart Meter case study is based on industrial requirements and evaluates the expressiveness of the higher-level composition tactics to design secure systems. There are several commonly-used combinations of primitives used in the case study, which I refer to as higher-level tactics. This case study shows that my approach can help design a secure system and that the higher level tactics are feasible to be applied. Furthermore, the case studies shows that the approach is applicable in different domains.

1.3 Research Method

This section explains the research approach adopted in this thesis. Figure 1.1 illustrates the steps undertaken as a sequence of the major research activities. Although this reflects the overall approach, each step may have had multiple iterations. For instance, when searching for existing security patterns (Step 1 and Step 2), I performed several iterations of literature search to collect patterns into a catalog.

Step 3 was the development of the concept of a design fragment that can be specialized to a particular platform. A design fragment is a partial realization of a design pattern in the context of a particular platform. In this step, I formalise security patterns' realisation for capability-based secure platform into reusable design fragments. This research activity addresses my first research question (RQ-1).

Step 5 was the development of a pattern-based composition approach to incremen-

tally build and verify an application design using platform-specific design fragments. This approach uses six primitive tactics. These six tactics were then proven by induction to preserve confidentiality. This research activity addresses RQ-3.

Steps 4 and 6 investigate the assurances of design fragments and application designs. This research activity has resulted in the concept of design fragment verification procedure. Besides verifying individual design fragments, I also verify the overall application design to provide assurance that its security goals are satisfied. I perform design-level verification as a step to reduce the cost of potential subsequent implementation verification. In verifying the application design, I reuse the verification procedures that are associated with design fragments in order to reduce the overall effort. The use of a verification procedure for a design fragment helps identify localized problems, and the reuse of that procedure for the application design helps to ensure that the overall application achieves its security goals. I embed these steps inside a security assurance case that allows for different verification techniques for different aspects of the design. The result of verification, performed using the verification procedures of the design fragments, provides evidence for one or more security claims in the assurance case. This research activity addresses RQ-2 and (partially) RQ-3.

Step 7 evaluates the composition approach using two case studies from different domain, a Continuous Deployment (CD) pipeline and a Smart Meter. The Smart Meter case study is based on industrial requirements

In this thesis, I am not concerned with the implementation of a design, but rather with the construction and assurance of the design. Designs can be thought of as formal specifications in the usual software engineering sense. This means that the implementation also need to correctly implement the design.

This approach helps to bridge the gap between security patterns and the underlying platform by providing platform-specific design fragments. A design fragment is repre-

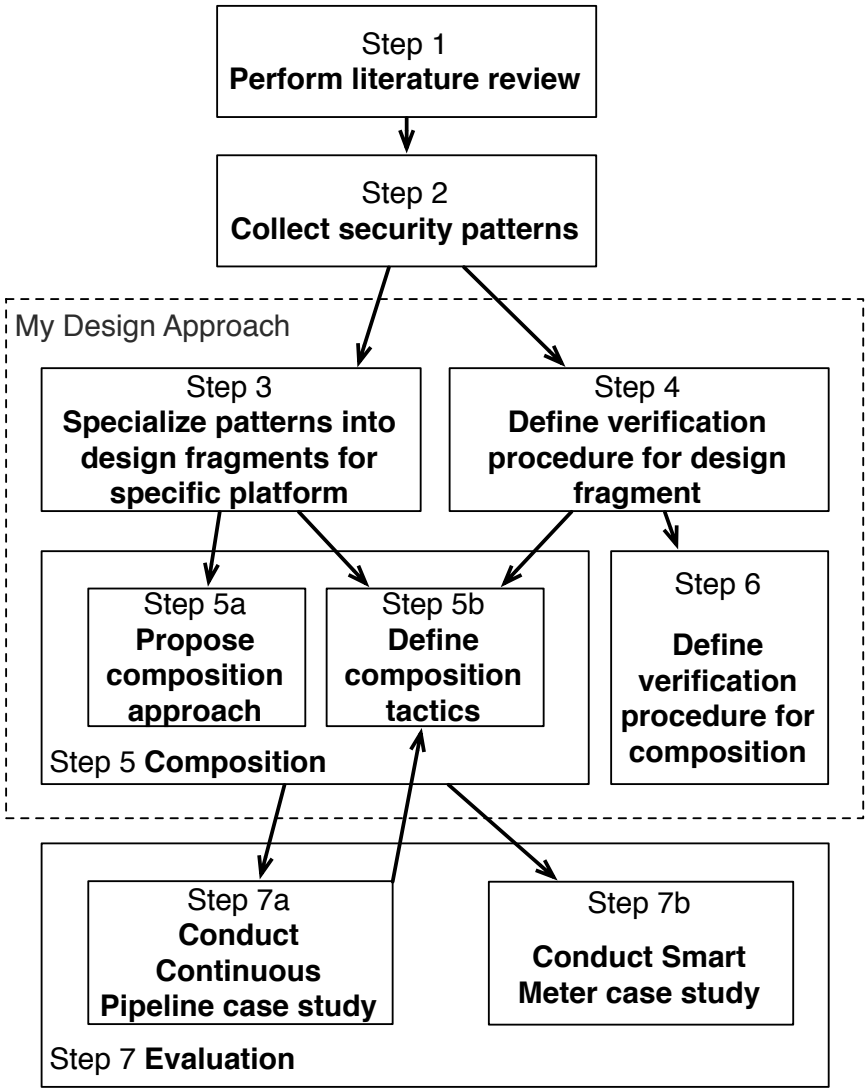


Figure 1.1: Overview of my research method

sented by a model that allows for design-level verification. This model consists of a component specification, which includes each component's behaviors, and the interactions between components in the model. Design fragments reduce the required step of manually translating the resulting application design to a specific platform for implementation purposes. Each design fragment is associated with reusable design-level verification procedures, which are used to verify that the design fragment satisfies the desired security properties. These security properties capture the informal security-related claim of the

security pattern. When a design fragment is selected and composed with an existing application design or another design fragment, the key challenge is specifying the composition. I tackle this through the identification and specification of pattern-composition primitives, which I call composition tactics.

My design fragments currently target platforms that adhere to the capability-based security model (Dennis and Van Horn, 1966). The general idea should be applicable to other kinds of platform, but exploring this is future work.

1.4 Thesis Organization

Chapter 2 provides a literature review of three bodies of work that are related to this thesis. The first is capability-based security and capability-based systems. The second is the area of security patterns. I start with the literature on organization and recognition of security patterns. Then, I review selection methods for security patterns and existing work on the verification of security patterns. I end by examining the application of security patterns to support security requirements. Finally, I review existing work on assurance cases.

Chapter 3 presents a security pattern template to capture existing security patterns, which are obtained from searching existing literature on the topic of security patterns. A search strategy that is inspired by systematic literature review is presented in this chapter. I synthesize a collection of 279 security patterns from existing literature. Finally, an analysis on the metadata of the pattern catalog is presented in this chapter.

Chapter 4 proposes the general concept of a design fragment, which is a partial instantiation of a design pattern for a particular platform. This thesis focuses on platforms that adhere to the capability-based security model. Capability-based design fragments are instantiations of security patterns for capability-based platforms which allow for for-

mal verification. A capability-based design fragment can be represented in textual and graphical forms. A methodology to derive a capability-based design fragment from a security pattern is also presented in this chapter.

Chapter 5 proposes a composition approach to build secure systems using six composition primitives (tactics). These primitives are proven by structural induction to preserve a particular security property, *Protected By*. This proof provides a basis to define more complex property-preserving tactics based on the six primitives. The definition of the *Protected By* property is presented in this chapter.

Chapter 6 provides the concept of a design fragment verification procedure, and presents a template to express the intended security properties.

Chapter 7 evaluates my composition approach using two case studies of different domains. Section 7.1 evaluates the applicability of the composition approach and the expressiveness of the six composition primitives to harden an existing system and to extend its functionalities. Verified capability-based design fragments are applied to a generic Continuous Deployment (CD) pipeline using the composition primitives to harden the security of the pipeline. The pipeline is then verified to be secure in the presence of various attacks. The approach is shown to be feasible to be applied in the context of a CD pipeline and to secure the pipeline. Furthermore, the composition primitives are sufficient to express the intended compositions. Section 7.2 evaluates the effectiveness of the higher-level composition tactics to design a secure smart meter, based on industrial requirements. Starting with a simple model, the composition tactics are applied to compose capability-based design fragments with the model in order to be withstand different attacks. The approach is shown to be feasible to be applied in a different domain and that the composition tactics are sufficient to express the necessary compositions.

Chapter 8 describes one possible solution to transform a verified application design into runnable code. I transform the design into a component-based framework specifi-

cation and require several pieces of additional information to make the code executable. This chapter provides evidence that it is feasible to implement our designs constructed using the approach proposed in this thesis.

Chapter 9 concludes and describes opportunities for future research.

Chapter 2

Background and related work

This chapter provides background on capability-based security, existing work in the area of security patterns and assurance cases. Section 2.1 provides an overview of capability-based security and existing capability-based systems. These systems include those that are distributed and those on a single machine. Section 2.2 reviews past and present literature in the area of security patterns. This starts with the literature on the organization of security patterns. Then, selection methods for security patterns are reviewed. I then analyze work on the verification of security patterns and examine the application of security patterns to support security requirements. Section 2.3 provides background on assurance cases.

2.1 Capabilities

Dennis and Van Horn (1966) introduced the concept of capabilities. A capability can be defined as "a token, ticket or key that gives the possessor permission to access an entity or object in a computer system" (Levy, 1984, p. 3). A simple analogy is with a key and locked doors. A valid key (i.e. capability) is needed to unlock the door in order to access the resource behind the locked door. A key can also be given to and redistributed by a

person.

Designing and building secure systems can be very difficult. Realisation of security relies on the features and mechanisms provided by the underlying system/platform. A capability-based system has several advantages. It offers a fine-grained control of access rights, which helps solve the Confused Deputy problem. The Confused Deputy Problem (Hardy, 1988) happens when a program which has permissions given to it for one purpose applies those permissions for some other purpose that is contrary to the original intent of the permission, and therefore allows something that it should not allow. A classic example (Hardy, 1988) of this problem involved a program that is allowed to write into a directory, which contains a log file and billing information file. The program takes a parameter of a file to which it will write debugging information. A user can then supply the billing information file into the program and thus overwriting the billing information. This may not have been intended during system design, but if the program has the necessary permissions, it may perform this action, perhaps under malicious or erroneous user control.

It is also easier to apply the Principle of Least Privilege (Saltzer and Schroeder, 1975) with a capability system. This principle requires that a user be given no more privilege than is necessary to perform a job. The system designer must determine the minimum set of privileges required for that user to perform a particular job and grant that user only those privileges. When a particular job is completed and no longer needed to be performed by the user, the privileges will be to be removed from the user. Removing a privilege is referred to as capability revocation.

Capability can be propagated or passed around. The Confinement Problem (Lampson, 1973) occurs when there is a leakage of capability, which results from inability to restrict capability propagation.

2.1.1 Capability Implementation Schemes

Capability systems have been used as an the underlying platform for building secure systems (Klein et al., 2009; Mullender et al., 1990; Shapiro et al., 1999). One of the earliest systems is the Cambridge CAP (Levy, 1984) in the 1970s. These systems have different implementation schemes. There are four different capability implementation schemes: tagged, partitioned, sparse and password. These are described below.

Tagged

In the tagged capability implementation scheme, Tags identify capabilities, which are used like normal pointers. Special hardware checks permissions on dereferencing capabilities and provides fast validation of capabilities. User code can copy capabilities. When a modification occurs, the modification turns the tag off, reverting capabilities to plain data. The tag can only be turned on by privileged instructions used by the operating system to make new capabilities. Distribution of capabilities is done by copying. A few shortcomings of this scheme are that capability hardware is complex and revocation is difficult to achieve due to inability to scan memory address space of other processes. The Cambridge CAP (Levy, 1984) was built using the tagged capability scheme.

Partitioned

In the partitioned capability implementation scheme, the operating system stores capabilities in a capability list. User code uses indirect references to the capabilities, i.e. a capability list index. The kernel is responsible for creating new capabilities, removing capabilities from capability lists and copying capabilities between different capability lists. Revocation can be supported easily as the kernel maintains the capability list. Systems that implement this scheme include EROS (Shapiro and Hardy, 2002; Shapiro et al., 1999), KeyKOS (Hardy, 1985), and seL4 (Klein et al., 2009; Sewell et al., 2011).

Sparse

Mullender and Tanenbaum (1986) proposed the scheme of sparse capabilities in 1986. In this scheme, a capability is typically made up of different pieces of information, each of which is represented in a fixed number of bits. A capability is stored as normal data. A sparse capability provides a probabilistic security, similar to an encryption. A bit-string is added to references to make valid capabilities a very small subset of the capability space. In this scheme, capabilities are pure user-level objects, which can be passed around like other data. The main protection is the sparseness of the capability space, which is large. Due to this sparseness, exhaustive search of the capability space by an attacker is infeasible. Encryption can be used as a second-line of defense to prevent unauthorized use and forgery of valid capabilities. Revocation can be done in this scheme by changing one of the bits. Amoeba (Mullender et al., 1990; Mullender and Tanenbaum, 1986) implemented a sparse capability scheme.

Password

In this scheme, a capability is made up of different pieces of information, each of which is represented in a fixed number of bits. One of the pieces of information that make up a capability is a password. A password is a large random number that designate access to an object. The size of the password is the main protection of this scheme and thus this scheme provides a probablisitic security of its capabilities, similar to sparse capabilities. A capability is stored as normal data. Password capability scheme is a flavour of the sparse capability scheme described above. The Monash password capability system (Anderson et al., 1986) was built using the password capability scheme.

2.1.2 Distributed Capability-based Systems

In this section, existing distributed capability-based systems are reviewed. Four of the systems are reviewed in great depth to identify the following: 1) capability representation; 2) capability protection mechanism; 3) capability revocation mechanism. The aim is to conclude that there is an abstract model of capabilities to represent most of the capability-based systems.

Amoeba

Amoeba (Mullender et al., 1990; Mullender and Tanenbaum, 1986) is a distributed capability-based operating system, which implements the sparse capabilities scheme. There are four main components of the hardware architecture: workstations, processor pools, specialized servers and gateways. Workstations allow users to access the Amoeba system. The processor pool provides the computing power, which can be dynamically allocated to the Amoeba system and users. Specialized servers distribute the processes to the compute power. Finally, gateways connect to other Amoeba Systems over wide area networks.

The software architecture is a client-server architecture. Its basic components are processes, messages and ports. Processes are active entities, communicating with one another by exchanging messages through their ports. Each of the objects in Amoeba is identified and protected by a capability. As shown in Figure 2.1, a capability consists of four different pieces of information: Port ID (48bits), Object ID (24bits), Access Rights (8 bits) and Check field (48 bits). The Port ID identifies the server. Knowledge of the Port ID implies send rights.

All communication in Amoeba starts with a client sending a request to a server. Clients send requests to server processes to carry out operations on objects. The server then accepts the request, does the work and replies back to the client. Operations in

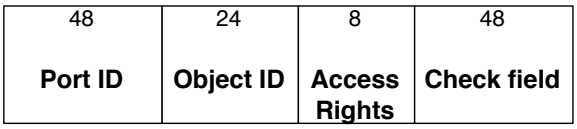


Figure 2.1: Amoeba Capability Structure

Amoeba are implemented by making remote procedure calls.

The key protection mechanism in Amoeba is to keep capabilities secret by embedding them in a huge address space. Amoeba uses the sparseness of the address space as its main protection mechanism. The knowledge of a port number provides users with access rights to a particular service in Amoeba.

Amoeba makes no guarantees about message delivery. It does not provide acknowledgments to the sending process. Furthermore, it is impossible to differentiate which bits are data and capabilities in Amoeba. This makes it difficult to implement revocation and leads to the Confinement Problem. The system will not know if capabilities are leaked.

Monash Password Capability System

The Monash Password Capability System (Anderson et al., 1986; Castro et al., 2008) was the first capability system to implement the password capability scheme. In this system, a capability is represented as a 128-bit binary value as shown in Figure 2.2. The first 64 bits contain the name of the object while the remaining 64 bits contain the access rights, which is represented by a password, to the object. Capabilities pointing to the same object with the same rights will have the same first 64 bits but will have distinct passwords (i.e. distinct last 64 bits).

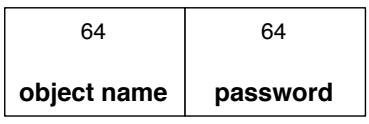


Figure 2.2: Password Capability Structure

There are two types of capabilities in this system: master and derived. A master capability is a capability that is created and given to the creator of an object. The creator specifies the rights for the master capability. A derived capability is a capability that is created either based on a master capability or an extant derived capability. A derived capability inherits all the rights of the capability it derives from. If a capability is destroyed, all capabilities that are derived from it are no longer valid, i.e. allows no access to an object in the system.

The Monash Password Capability System cannot provide absolute guarantee of security, i.e. it provides probabilistic security. In this system, any 128-bit value may be a valid capability. The authors argue that the probability of guessing a valid 64-bit password is very low, which is 10^{-15}. To prevent more optimized password guessing attack, the system generates a password using a random value obtained from a thermal noise peripheral.

The authors introduces the concept of "money" to dispose garbage, which are objects with no remaining owner capabilities. The rent collector (essentially a garbage collector) periodically scans the master capability for each object and deducts rent from its money. If the object has insufficient money, it will be regarded as garbage and destroyed.

A limitation of the Monash Password Capability System is that it cannot differentiate between capabilities and data. This leads to the Confinement Problem. Furthermore, the probability of 10^{-15} to guess a valid password was considered to be safe in the 1980s but is now not recommended by NIST (Barker et al., 2007). NIST recommends at least a 128-bit key.

Annex System

The Annex system (Grove et al., 2007) is a distributed capability-based system that implements a password capability scheme. It extends the partitioned capability scheme

with the password capability scheme. The capabilities are stored as regular data but resides within the protective bound of the kernel. Objects outside the kernel invokes the capabilities through *handles*, which are created by the kernel when an object receives a new capability. This capability is then swapped with the newly created handle, transparent to the object. The kernel keeps a mapping between the capability and the handle.

A capability in the Annex system is a 384-bit value. Figure 2.3 shows the structure of an Annex capability. The first 64 bits contain information about the host of the object (*deviceID*). The next 48 bits represent the identifier of the object in its host (*objectID*). Then, 16 bits are reserved for a capability identifier which is assigned when a capability is created. This is used to differentiate capabilities that refer to the same object but have different rights. The last 256 bits contain the access rights.

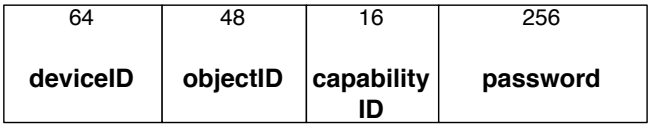


Figure 2.3: Annex Capability Structure

The authors argue that revocation is the only way to invalidate a capability and that the challenge is to determine who holds a capability. They developed a kernel-based algorithm to keep track of the capability propagations that happen in each device. These propagations are stored in a graph. When a particular capability needs to be revoked, the device will be notified to remove the capability from its graph. Then, a recursive flushing mechanism is used to delete all copies of that capability from all devices.

CapAuth

CapAuth (Cai et al., 2010) is a capability-based handover scheme for mobile towers, which aims to minimize latency and network traffic. It implements the sparse capability scheme.

CapAuth allows users to play an enhanced role by acquiring signed capabilities (signed assertions of context) from access points. During a handover, users ensure that their new access point has the necessary authenticated context by providing their capabilities. The use of capabilities allows fast and simple context transfer as it involves only a simple message exchange to transfer a capability from a user to an access point.

In CapAuth, a user only needs a valid capability to authenticate to an access point. CapAuth implements a mechanism to limit the usage of a capability to only once. This prevents potential problems when a malicious user clones his capability to access services simultaneously at multiple access points.

In order to support revocation of capability, CapAuth includes a time-to-live (expiration) in the capability. A centralized server is then responsible for re-issuing capabilities.

Summary

Distributed capability based systems generally follow either the sparse or password implementation scheme. They share a common problem, which is the Confinement Problem. This is when the system cannot limit or prevent access leakage (i.e. of capabilities). Both sparse and password schemes cannot differentiate between capabilities and data. Therefore, there is no way to prevent access leakage.

The systems reviewed above contribute to the idea that there exists a general capability model. Firstly, there are similarities of information in capability representation. For instance, the ObjectID in Amoeba and Object Name in Monash Password Capability. Secondly, both schemes depend on sparseness for their protection. Sparse scheme depends on the sparseness of the address space (port numbers in Amoeba) and password scheme depends on the sparseness of the valid password values.

2.1.3 Single Machine Capability-based Systems

There are capability-based systems that are targeted for single machines. The Cambridge CAP computer (Levy, 1984), developed in 1970s, was the first system to demonstrate the use of capabilities for security. Capabilities were implemented in hardware. KeyKOS (Hardy, 1985) is a capability-based operating system. Capabilities are protected by the kernel (partitioned scheme). KeyKOS has influenced the design of EROS (Shapiro and Hardy, 2002; Shapiro et al., 1999), and in turn seL4 (Klein et al., 2009; Sewell et al., 2011). The aim of this review is to conclude that there is an abstract model of capabilities to represent most of the capability-based systems.

KeyKOS

KeyKOS (Hardy, 1985) is a capability-based operating system that implements the partitioned capability scheme. In KeyKOS, a capability is referred to as a key. Entities in KeyKOS communicate with each other by sending a message via a key. A key to the recipient is required before messages can be sent to it. Sending a message to an entity using a key means invoking a function of the receiving object, together with the intended parameters. Using a key is to be seen as invoking a function while the accompanying message contains the parameters for that function invocation. As an entity can be referred to by many keys, each key contains an 8-bit field that is used by the recipient to identify the sender. Note that if an entity, *Alice*, does not own a key to another entity, *Bob*, *Alice* cannot communicate with *Bob*.

There are three ways for invoking a key: *Fork*, *Call* and *Return*. *Fork* sends a message without waiting for the response, while *Call* waits for the response. *Return* is used to send back response by using the message's resume key, which is created when *Call* is used to send the initial message.

In KeyKOS, only the kernel that has access to keys and creating a key to an entity is

a privileged operation. An entity is allowed to duplicate keys that it holds. A key is not associated with read or write rights but it signifies that the holder is allowed to invoke a particular function of the recipient.

EROS

EROS (Shapiro and Hardy, 2002; Shapiro et al., 1999) is a capability-based kernel that is inspired by KeyKOS. It implements the partitioned capability scheme, which means that it is secure through kernel protection. A capability in EROS is represented as a pair that contains an object identifier and a set of permissions. The authors claim that the capabilities are unforgeable and tamper proof. Each objects and its capabilities has a version number. The capability is valid (i.e. convey authority) if both the version number of the object and its capabilities match.

All resource accesses are performed by invoking capabilities. Capability invocation is the only way to perform function calls in EROS and each invocation is checked by the kernel.

EROS uses two mechanisms to limit access propagation: weak access rights and access indirection. A weak access right is a read-only right. Any capabilities that are obtained using a weak access right will always be granted a read-only access and are tagged as "weak". This ensures transitive read-only access and limits access propagation. Access indirection is a mechanism whereby an indirection object is supplied in place of the real capability. The access can then be revoked by destroying the indirection object.

seL4

seL4 (Klein et al., 2009; Sewell et al., 2011) is a formally verified microkernel that draws inspiration from EROS. It implements the partitioned capability scheme, which means that it is secure through kernel protection. It is the first microkernel whose source code

has been formally verified for functional correctness, i.e. its implementation always follows an abstract high-level specification of the kernel behavior. This provides high assurance that the kernel will never perform an unsafe operation. The proof relied on assumptions that some assembly code, boot code, management of caches and hardware are correct.

In seL4, capabilities are associated with access rights. The possible access rights are Read, Write and Grant. A Read right allows the holder to read or receive data from the object. A Write right allows the holder to write or sent data to the object. A Grant right allows the holder to create new capabilities that refer to the same object but have different properties. All newly created capabilities have all the possible access rights, but these can be reduced by the MINT operation. This produces a new capability from an existing capability with the same or fewer access rights. This operation can also be used to add "tags" to a capability. A tag is used to identify the invoker of an object.

Access propagation can be limited in seL4 by deleting any capabilities using DELETE operation. REVOKE operation deletes each child of the specified capability but not the capability itself.

Cambridge CAP

The Cambridge CAP computer (Levy, 1984) used a capability-based operating system and hardware architecture. It implemented the tagged capability scheme.

The hardware architecture of the CAP computer consisted of a microprograming control unit, micro-control storage and arithmetic unit. The CPU kept evaluated capabilities (capabilities and the primary location they address) in a 64-entry capability unit (Levy, 1984).

The CAP system provided a process tree structure, as illustrated in Figure 2.4. A master coordinator, which controls all system hardware resources, is at the root of the

process tree (level-1). The master coordinator allocates resources to its subprocesses (level-2). A level-2 subprocess can also create further subprocesses.

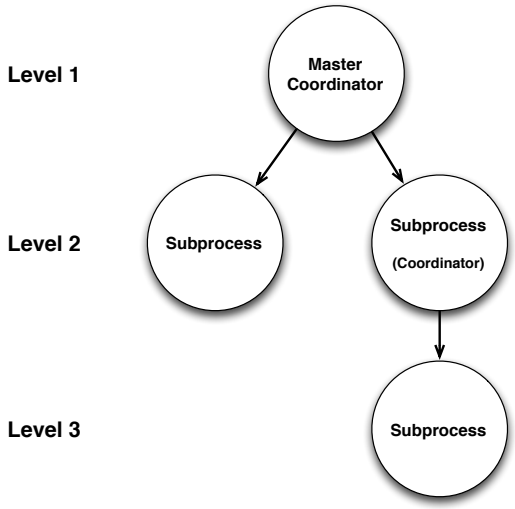


Figure 2.4: Cambridge CAP Process Hierarchy. (redrawn based on (Levy, 1984))

Each capability contains a type field in the two high-order bits which differentiates different capability types. As shown in Figure 2.5, bits marked W and U are set by hardware to indicate that a segment has been written or accessed respectively. There are five access rights in the CAP system: write capability (WC), read capability (RC), read data, write data and execute data.

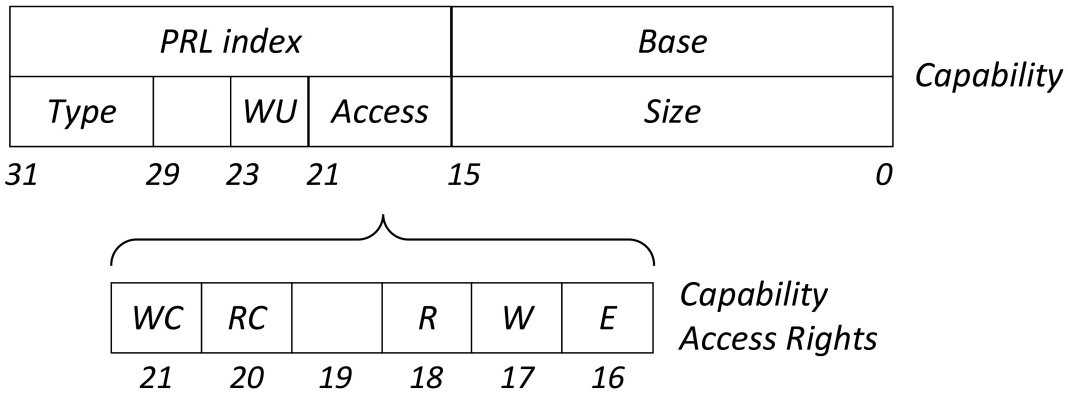


Figure 2.5: Cambridge CAP Capability and Access Rights Format (redrawn based on (Levy, 1984))

Summary

Single machine capability systems are restrictive in terms of sharing capabilities, as capabilities live inside the protective bounds of the kernel. There is a trade-off between restrictiveness, which means degree of ability to share capabilities freely, and security. In distributed capability systems, which generally follow either the sparse or password capability schemes, it is easier to share a capability but it is almost impossible to restrict capability propagation and thus to control leakage of capabilities (the Confinement Problem). The Confinement Problem is not an issue with single machine capability systems that implemented the partitioned scheme, as the creation of every capability needs the intervention of the kernel. In a system that implements the tagged scheme, the Confinement Problem occurs as capability propagation cannot be restricted and revocation is infeasible.

The partitioned systems reviewed have similar capability representations, which contributes to the goal of having a general capability model. This is due to one drawing inspiration from the other.

2.1.4 Serscis Access Modeller (SAM)

Serscis Access Modeller (SAM) (Leonard et al., 2013) is a modelling tool and notation for defining and verifying capability-based systems. A SAM model represents the distribution of capabilities across the components that make up a capability-based system. SAM has a textual representation and a graphic representation. The textual representation consists of three specifications, namely component, initial capability distribution, and security goal. In the component specification, components are represented as classes defined in a Java-like language. The behaviors of each component are represented as functions. The key characteristic of the behavior that we are interested in is the invocation of other components and the capability propagation through those invocations. The

initial capability distribution specification defines the capabilities held by each component at the start of the system's execution. The security goal specification consists of Datalog (Ceri et al., 1989) rules and queries that are used to verify the system.

The graphical representation of SAM shows the state of the system/model where all the possible capabilities have been propagated and shows whether there is any security violation. A model is a directed graph with nodes representing components and directed edges (arrows) representing access. An arrow pointing to a component represents holding a capability to it. Holding a capability to a component in SAM provides all the permissions, i.e. read, write, invoke and grant, to that component. In SAM notation, an arrow has different styles (dotted and solid) and different colors (green and black). A solid arrow signifies that the capability is obtained as part of the initial capability distribution, while a dotted arrow means that it is obtained at runtime. This is useful for tracing how access has been passed from one component to another. Consider a system with two components *Alice* and *Bob*. A green arrow from *Alice* to *Bob* means that *Alice* has invoked functions of *Bob*, while a black arrow indicates *Alice* does not invoke functions of *Bob*. Furthermore, components can be trusted (black) or untrusted (blue). Trusted components have assumptions about their behavior (i.e. they will only call methods as instructed) while untrusted components may call any method of the components that they have access to. In addition, untrusted components may try to pass around any capabilities they possess to the components they can access. Note that a SAM model defines the upper bounds on the behaviors of the trusted components and is an over-approximation of the real system.

One of the limitations of SAM is that it cannot model timing properties. SAM can be used to verify that an attacker does not invoke a function in a component. However, it cannot verify that the attacker will not learn about the content of a file via timing attacks.

The second limitation is that SAM cannot model dynamic deletion. Once the model

is executed, a component cannot be removed. Deletion of a component can be done during design time (i.e. before the model is executed). SAM does, however, allow dynamic creation of components.

2.1.5 Conclusions

Most capability systems follow the object capability model (Miller, 2006). All of the single machine capability-based systems that are reviewed follow the object capability model. As for the distributed capability-based systems, the Annex system is an example of a distributed object capability model. Object capability models can be paired with password capabilities to support distributed capability computing. There are similarities between the different capability concepts, even though they are implemented differently. The similarities are that: 1) capabilities are transferable and are one-directional; and 2) access rights, represented by capabilities, are corroborated by the principals/actors. Therefore, I conclude that there is a general capability model to represent most of the capability-based systems. As introduced earlier, Serscis Access Modeller (SAM) (Leonard et al., 2013) is a modeling tool for capability-based systems that follows the object capability model and thus will be used to model designs in this thesis.

The difference in capability representations matters a great deal in implementation. At the design level, one will care about whether this component has read capability/rights to a component, without much focus spent on how these capabilities are represented. However, the differences between capability implementation schemes sometimes do affect designs. This is true when it comes to limiting capability propagation, which is harder to limit with sparse or password capability systems than with the partitioned scheme. On the other hand, a partitioned scheme might not be easy to implement for a distributed capability system. It may not be possible for a design to be readily implemented under all capability schemes. However, understanding different capability

schemes and capability representations, we will be better able to judge which scheme is more suitable to realize a given system design.

2.2 Security Patterns

This section reviews literature in the area of security patterns. It is divided into two main parts. Section 2.2.1 reviews literature on organization and recognition of security patterns to aid designing a secure system. Section 2.2.2 assesses the existing selection method of security patterns. Section 2.2.3 analyzes work that have been done on the verification of security patterns. Application of security patterns to support security requirements is discussed in Section 2.2.4.

2.2.1 Organization and Recognition of Patterns

A pattern is an encapsulation of a solution to a recurring problem (Alexander et al., 1977). A pattern has four essential elements, including *pattern name*, *problem*, *solution* and *consequences* (Gamma et al., 1995). The *pattern name* conveys the essence of the pattern concisely. The *problem* field describes when to apply the pattern together with its explanation and context. The *solution* field identifies the elements that make up the pattern and their relationships. The *consequences* field shows the trade-offs and results of using the pattern. The other fields in their template are: *intent*, *also known as*, *motivation*, *applicability*, *structure*, *participants*, *collaborations*, *implementation*, *sample code*, *known uses*, and *related pattern*. The *also known as* field captures aliases of a particular pattern. As different authors have proposed different security patterns, some patterns use different names for the same information. The *motivation* field illustrates a scenario, which contains design problem, and shows how the pattern solve the problem. In addition, the *related patterns* field lists other patterns that are closely related to the

pattern and outlines their differences. The *known uses* field describes examples of the pattern used in real systems.

The template that is proposed by Gamma et al. (1995) is a baseline that has been adapted for security patterns. Yoder and Barcalow (1997) used fields from Gamma et al. (1995), namely *pattern name*, *also known as*, *motivation*, *known uses*, *related patterns* and *consequences*. Yoder and Barcalow (1997) split the *known uses* field into two: *known uses* and *non-security known uses*, and introduce two new fields, *forces* and *examples*, to enrich the description of a security pattern. The *forces* field highlights the impact of the pattern and the *examples* field demonstrates the pattern in use.

Konrad et al. (2003) incorporated a new field, *supported principles*, into their pattern template. This field details which of the ten security principles, proposed by Viega and McGraw (2011), are supported by the pattern. These principles are fundamental to the development of secure systems. These ten principles are "securing the weakest link", "defense in depth", "secure failure", "least privilege", "compartmentalization", "simplicity", "promote privacy", "it's hard to hide secrets", "don't extend trust easily", and "trust the community". They also added the *behavior* field, which consists of UML sequence and state diagrams, to represent the behavioral aspect of a pattern, and, the *constraints* field, which contains global conditions that needs to be upheld for the security pattern to achieve its intended goal. The authors adopted the idea of classifying patterns into the categories from Gamma et al. (1995). However, this work does not consider the relationship between security patterns and an underlying platform.

VanHilst et al. (2009) proposed a classification of security patterns based on a multi-dimensional matrix of concerns, where each dimension represents a distinct list of concerns. One important dimension proposed is the Software Lifecycle Stage. The stages are defined as follows: Domain Analysis, Requirements, Problem Analysis, Design, Implementation, Integration, Deployment, Operation, Maintenance and Disposal. This

dimension allows software architect in selecting the appropriate patterns for the right development stage of the software. Another interesting dimension is Architecture Layer. The Architecture Layer consists of Client, Logic, Data, Operating System, Distribution, Transport and Network Layer. This dimension enables software architect to filter patterns based on different layers of the architecture (VanHilst et al., 2009).

Alvi and Zulkernine (2011) proposed a pattern classification scheme based on security flaws. Their classification scheme is associated with software development lifecycle, in particular the requirement stage, design stage and implementation stage. They propose different parameters for each stage: security flaws and security objectives for the requirement stage, security flaws and security properties for design stage and security flaws and attack patterns for the implementation stage.

Based on their proposed classification parameters, they added fields to their security pattern template for *security objectives and properties*, *related security flaws* and *related attack patterns*. The *security objectives and properties* field contains the main objectives and properties of the security pattern, which are essential for solving problem. The *related security flaws* field lists all the related security flaws from a public software flaws database called Common Weakness Enumeration (CWE)[1]. The *related attack patterns* field identifies the related attacks and attack patterns to the pattern. This classification aims to allow developers to filter security patterns based on a security flaw.

Hafiz et al. (2007) proposed a new classification schema based on different dimensions: fundamental concepts of security, application context, Zachman framework (Zachman et al., 1987) and Microsoft STRIDE threat model (Swiderski and Snyder, 2004). The fundamental concepts of security, proposed by Avizienis et al. (2004), categorized security patterns into the security issues addressed (i.e. Confidentiality, Integrity and Availability). The application context classifies patterns according to the part of the

[1]https://cwe.mitre.org/

system that they are securing, which are *core*, *perimeter* and *exterior*. The *core* patterns consider the security mechanisms inside a system. The *perimeter* patterns consider the authentication and authorization issues while the *exterior* patterns are concerned with data transmission. The Zachman framework is used to classify the patterns based on stakeholders and concerns. The STRIDE threat model classifies security threats into six categories: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

Laverdiere et al. (2006) surveyed and evaluated a number of security patterns catalogs and discovered that there is a recurring issue with the patterns: most manifest some undesirable characteristics. These characteristics are over-specification, under-specification, lack of generality, lack of consensus and misrepresentation. Over-specification means that the specification provides too much information and properties than needed or has some variants. According to the authors, having some variants could create confusion, especially in determining which variants to use and thus is considered as over-specification. Under-specification is when the pattern does not have sufficient properties or is incomplete. There are four levels of abstraction for patterns (Kienzle and Elder, 2002), namely Concepts, Classes of patterns, Patterns and Examples. Lack of generality means that the pattern claims to be at different level of abstractions than it is in actuality. This means that it is not general enough to be applied in multiple contexts. Lack of consensus occurs when a pattern has a different concept than its name, or when either the intent or solution of a pattern is not uniform across the literature. Misrepresentation of a pattern occurs when a pattern's name or intent is inconsistent with the other properties of the pattern. Based on these undesirable characteristics, the authors proposed a *trade-off* field for security patterns. This field explains both the positive and negative effect the pattern can have on quality attributes. It can help architects when applying the pattern to a system, as he/she is well informed of the impact that the pattern

has.

Rosado et al. (2006) presented a framework to compare patterns. The comparison is based on predefined criteria for the most commonly used security attributes. These include confidentiality, integrity, availability, authentication, authorization and maintainability among others. They also included some evaluative criteria: performance, implementation costs and security degree. Performance and implementation costs indicate the impact of the pattern on the performance of the system and the cost of implementing it respectively. The number of security properties covered determines the degree of security. A higher security degree signifies that more properties are covered by the pattern. However, Rosado et al. (2006) did not mention how the comparison is done nor the technique that they use.

Washizaki et al. (2009) defined two new types of models, Dimension Graph and Patterns Graph, to improve the classification of patterns. The Dimension Graph relates patterns to predefined dimensions, based on the work of VanHilst et al. (2009). These dimensions are Software Lifecycle Stage, Architectural level, Concern, Type of Pattern, Domain and Constrain. The Dimension Graph is useful in exploration and analysis of possible dimensions for target patterns. The Patterns Graph shows the relationship between different patterns. Relationships can be either Association, Generalization or Aggregation. Association relationships are where the pattern provides or receives some features from other patterns, while Generalization and Aggregation signify that the pattern is abstract or is composed of other patterns respectively.

A limitation with this work is that only two patterns, RBAC and Authorization, were used as examples in the evaluation of the framework. Although the authors focus on relationships between patterns, they do not relate security patterns to underlying implementation platforms.

Summary

Many authors have investigated recognizing patterns and organizing patterns. This includes enriching security patterns with additional information. Classification of patterns, which groups patterns into small correlated sets, is an aspect of pattern organization (Hafiz et al., 2007). However, most research has not considered the relationship between security patterns and underlying platforms. This leaves a gap. Furthermore, some patterns are underspecified, as pointed out by Laverdiere et al. (2006). This might result in patterns not fulfilling their aim when being implemented, as developers need to make assumptions and personal interpretation of the patterns.

2.2.2 Selection of Patterns

There have been a large number of security patterns proposed, so selecting the appropriate pattern can be a difficult task (Hafiz et al., 2012). Weiss and Mouratidis (2008) attempt to mitigate this problem by presenting a formalized approach to select security patterns and to reason that the selected patterns satisfy security requirements. The authors implemented a search engine to search for security patterns based on security requirements. Goal-oriented Requirements Language (GRL) is used to reason about requirements and Prolog is used to reason about the type and strength of contributions made by each pattern to security-related Non-Functional Requirements. Although this approach automates parts of the pattern selection process, it fails to consider security models in which the security patterns will be implemented. Moreover, the Prolog rules that authors have handcrafted have limitations. The types and strengths of the contributions a pattern provides are not precise. In their work, one of the possible strengths of contributions is unknown, which defined as indicating that there is a contribution from a pattern but the extent and sense of the contribution is unknown. Furthermore, the performance and accuracy of the approach has not been evaluated.

Hasheminejad and Jalili (2009) proposed an automatic selection approach for security patterns by using text classification and information retrieval techniques. This approach consists of two phases, namely Learning Classifiers of Security Patterns and Security Pattern Suggestion. In the Learning Classifier phase, the problem and context sections of security patterns are preprocessed, i.e. removing stop words and removing suffix using Porter's stemmer algorithm (Porter, 1997), before creating a training set for each security pattern class. Then, they choose 70% of the documents at random and evaluate the learned classifiers for the next phase. The Security Patterns Suggestion phase starts with preprocessing the security problem and then applies the weighting method selected in the learning classified. Then, the similarity is computed with a cosine similarity technique. A pattern is only recommended when the similarity between the security pattern vector and security problem vector is above the threshold.

This approach has a number of limitations. Firstly, it fails to consider the relationship between security patterns and security models. This is crucial as security patterns need to be built on top of a system, which adheres to a particular security model. Secondly, the approach does not consider other aspects of the patterns, such as relationships with other patterns. It will not help the composition of patterns just by considering only the problem and context of the security patterns. For example, Single Access Point pattern solves the problem of having multiple entry points to a system but it depends on the Check Point pattern to perform the necessary security checks. These limitations make the approach insufficient in assisting an architect to pick out the right set of security patterns.

Yskout et al. (2012) carried out an empirical study to investigate whether annotations on security patterns are beneficial. This experiment was done with 90 master students who were asked to a reinforce software architecture with security patterns and to complete a questionnaire after the experiments. The students were divided into pairs and half of them (22 pairs) were given the plain pattern catalog while the remaining 23 pairs

were provided with the annotated pattern catalog. The experiment evaluated metrics for selection time, selection efficiency and the number of irrelevant patterns discarded. The expected outcome was that the annotated group would perform the tasks more efficiently and quickly. Surprisingly, the annotated group did not complete the tasks faster than the other group. It did however operate more efficiently than the plain group. The result of the survey shows that the annotated group consider the relationship among patterns as helpful in selecting the appropriate pattern.

Summary

Selection of security patterns is an essential step (Heyman et al., 2007) in designing a secure system with security patterns. The large number of security patterns makes this a difficult task for software architects (Kubo et al., 2007). Most studies in the area of security patterns have not considered the relationships between different patterns, but this is important in assisting architects to compose security patterns to build systems.

2.2.3 Verification of Security Requirements with Patterns

Formal methods are mathematically-based techniques for the specification, modeling and verification of a system. They help to increase the assurance that the security requirements in a secure system are upheld by mathematically proving them (e.g. by means of model checking). A model checker does an exhaustive search of all possible states that a system can reach during its execution and provides any counterexamples found. Despite their potential contribution to the assurance of systems, these methods are not widely used software architecture design (Heyman et al., 2012).

Verification of security requirements in patterns

Preliminary work on verifying security requirement properties has been undertaken by Konrad et al. (2003). This extended previous work (McUmber and Cheng, 2001) on a framework and tool to formalize UML. UML diagrams are checked for consistency and transformed to Promela, a formal language, by Hydra (McUmber and Cheng, 2001). A state-of-the-art model checker, SPIN (Holzmann, 1997), is used on the generated Promela code to determine whether the properties are satisfied. Any counterexamples produced by SPIN are visualized in terms of UML diagrams by a visualization tool, Minerva. A revised model maybe passed on to Hydra and the process is repeated until no counterexample is generated by SPIN.

The work contributes a complete tool suite to formalize security requirements that are embedded inside a pattern. However, one possible issue with this approach is scalability, because of the possibility of state explosion. The case study used is a simple one. As the complexity of the system increases, the number of possible states to check grows exponentially. Furthermore, in the case study, a class represents a pattern. In a real system, a pattern can be applied to many classes and each class may require the composition of several patterns, i.e. require the information about the relationships between security patterns. This is missing from this work.

Heyman et al. (2012) presented an approach to formally verify that security requirements in a security pattern are satisfied. They model the pattern as an abstract model in Alloy (Jackson, 2012), a first order logic language for describing structures and verifying model properties. The Alloy analyzer will then verify the abstract model and generate counterexamples to check properties of the model. A counterexample may indicate that there is a flaw in the architecture that the security requirement is underspecified, or that there is some other limitations that the architect did not take into consideration. The abstract model can then be refined. The refined model is intended for a security expert

while the abstract model is targeted for software architect, as it is easier to understand.

The introduction of trust assumptions in a pattern is an interesting contribution of this work. Trust assumptions can be characterized as either expectations or residual goals. Expectations are assumptions that a component will behave in a certain way, while a residual goal describes the security concerns that need to be addressed but are not in the scope of the pattern. This concept of trust assumption is tightly coupled with a relationship between security models and security patterns. It assumes that there is a security model or system that enforces or supports the expectation.

One limitation of this approach is scalability. Furthermore, there are several concerns about the evaluation of this approach. Firstly, it uses a very small sample size, i.e. two participants. Secondly, both participants have vast experience in security patterns and software security. Evaluating an approach with security experts would taint the result of the experiment. The evaluation could be improved by experimenting with the approach using a larger number of participants who are less experienced in the area of software security.

Verification of security requirements in whole system with patterns

Dong et al. (2010) proposed an automated approach to verify the composition of security patterns by means of model checking. They specify the behavioral aspect of a security pattern, which is captured in a UML sequence diagram, in Calculus Communicating Systems (CCS). The behavioral aspect includes synchronous messages, asynchronous messages and alternative flows. They provide a guideline to specify the behavior in CCS and define different message types into CCS specification. They have also provided a proof that their CCS specification is faithful with respect to the behavioral model of patterns in a sequence diagram. A CCS-based model checker, CWB-NC, is used to perform the analysis and verify that the characteristics of each security pattern still hold

after their compositions.

A limitation of this approach is that a user needs to manually transform sequence diagrams to CCS using the proposed guidelines. This requires expertise in both security patterns and CCS. Furthermore, there might be scalability issues if the complexity of the system increases as this approach is using a model-checking technique. Finally, their guidelines do not include any information on transforming loops, which are common in real systems.

Summary

All of the previously mentioned methods suffer from some limitations. One common limitation is scalability which is a major concern for model checking. Furthermore, an underlying platform is required to realize security patterns. As mentioned in Heyman et al. (2012), expectation and trust assumption are required for the system. A relationship between patterns and security models is essential, but prior most studies in the area of security patterns have not focused on this.

2.2.4 Application of Patterns

Shiroma et al. (2010) proposed an automated application technique for security patterns in model driven software development. The software architect first manually defines the abstract model and then translates it into a more concrete model with model transformation rules. These rules, which are handcrafted by the architect, consist of a precondition, an argument and operation. The precondition varies according to the dependencies between patterns, i.e. a pattern P that is relied upon by another pattern Q becomes the precondition of that pattern Q. This dependence can be obtained from the Related Patterns field of the pattern. Some dependence is not stated clearly and therefore needs to be inferred from the patterns' Context and Problem field. The Argument is a parameter

that the user needs to provide. The Operation is the mapping of classes and relations between classes in the model during the application of the pattern. This approach checks for dependencies between patterns at each application of a pattern and produces a model. Then, a mark will be left in the model after it is transformed. This helps the architect to keep track of the existing application of patterns in the system.

Surprisingly, this work claims to saves 71% of time spent compared to manual application of the patterns. Unfortunately, it does not report on the accuracy of the application and whether the models satisfy security requirements. Furthermore, manual extraction of dependencies from the patterns leaves room for error and is not straightforward. Experience and ample time are required to deduce dependencies, especially from the Context and Problem field of patterns. A limitation of their study is that it only used four unique patterns. As mentioned by Hafiz et al. (2007) and Rosado et al. (2006), there are overlaps and variants of patterns and extracting dependencies from these variants of pattern will be difficult.

Mourad et al. (2010) put forward a pattern development and deployment approach based on Aspect Oriented Programming. This approach consists of two phases. The first phase requires security experts to devise security solutions, which consist of aspects and patterns, and give detailed information on how and where to integrate each pattern. Then, the second phase is performed by users by implementing aspects corresponding to the patterns and integrating them into the system. One major limitation of this approach is their first phase where security experts are required to draw up a solution and to detail the integration. Thus, there is no reusable knowledge for designing a secure system.

2.2.5 Composition of Patterns

Bayley and Zhu (2008) investigated composing design patterns. Each of the design patterns (Gamma et al., 1995) is defined in a formal specification in three parts: *component*,

static condition and *dynamic condition*. The *component* contains a set of variables. The *static condition* contains the structural part of the pattern, based on the class diagram for the pattern, presented in the Structure field in the Gamma et al. (1995) template. The *dynamic condition* contains behaviour of pattern based on sequence diagram, presented in the Collaboration field in the template. They have defined three operators: specialization, renaming components in pattern and overlaps. They then compose patterns using overlap operators. Specialization and renaming operators are not used for composition.

One of the limitations of their work is that their formal definition of the overlap operator and composition using overlaps are not precise (and not accurate). Their definition for composition states that given two patterns, P and Q, and a set of overlapping components, o, the result is the combination of a modified P, a modified Q, and the set of overlapping components o. However, they do not define what modifications need to be done on both P and Q. The steps for their composition: 1) identify an overlapping component between the two patterns; 2) create a new component; 3) combine this component with the two patterns. However, there are two implied steps of removing the overlapping component from each patterns before combining the patterns with the new component to form the composite design. These steps have not been defined.

Furthermore, the overlap operator is not sufficient to express other interesting compositions, i.e. composing two patterns without any overlapping components. For example, the Single access point pattern (Yoder and Barcalow, 1997) requires the Checkpoint pattern (Yoder and Barcalow, 1997) to perform the necessary security checks. However, there are no overlapping components between them and thus they cannot be composed with this operator.

Finally, they depend on the class diagram and sequence diagram for a pattern. While the pattern template proposed by Gamma et al. (1995) includes these diagrams, other pattern templates (Steel et al., 2005; Yoder and Barcalow, 1997) do not include them.

2.2.6 Conclusion

Security patterns can contribute to building secure software systems. They encapsulate accumulated knowledge and best practices of security experts' solutions to recurring security problems. As security patterns become more popular, their numbers grew rapidly and there were many overlaps. This can make selection of appropriate patterns more difficult for architects. Furthermore, the lack of information on the implementation of security patterns hindered its adoption within the industry (Ortiz et al., 2010).

Verification of security requirements for security patterns can raise the level of assurance when designing a secure system. Building a system on top of verified patterns is a step towards raising the assurance of a secure system. Formal verification techniques can be applied to the whole system to increase the assurance of the system. However, caution must be taken when applying formal methods as previous studies have shown that they are prone to scalability issues.

2.3 Assurance Cases

An assurance case is a structured argument that a system has certain properties (Bishop and Bloomfield, 1998). It provides confidence that a system will function as intended through evidence and reasoning (arguments) that link the pieces of evidence to the claims. There are three main elements in an assurance case (Graydon et al., 2007): claim, argument and evidence. A claim represents a desired security property that the system should achieve. Argument is an explanation of how the evidence supports the claim to be true. Evidence is a proof that the system has certain property and can be obtained through testing, analysis or verification.

There are two commonly used notations to represent an assurance case (Bloomfield and Bishop, 2010; Kelly, 1999), which are Claim-Argument-Evidence (CAE) and Goal

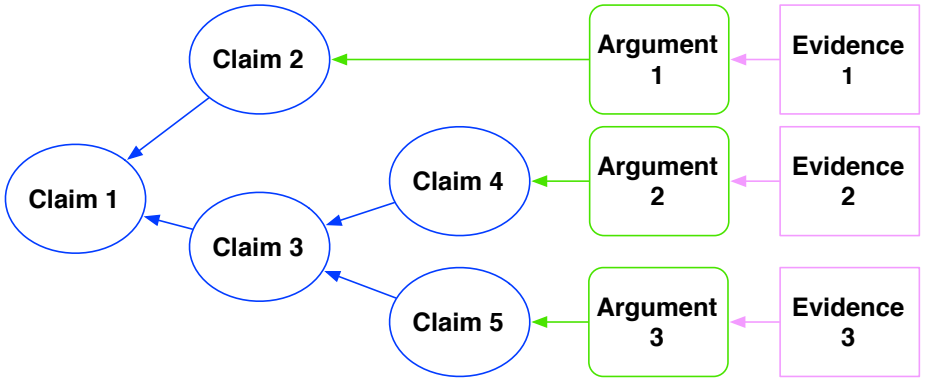


Figure 2.6: Assurance case with Claim-Argument-Evidence Notation

Structured Notation (GSN). The CAE notation has three elements, i.e. claim, argument and evidence. A claim can be decomposed into multiple subclaims. A claim is true and is supported by evidences if each of its subclaims is supported by an evidence. Figure 2.6 shows an example of an assurance case built with CAE notation. Claim 1 is true if Claim 2 and Claim 3, and by extension Claim 4 and Claim 5, are true.

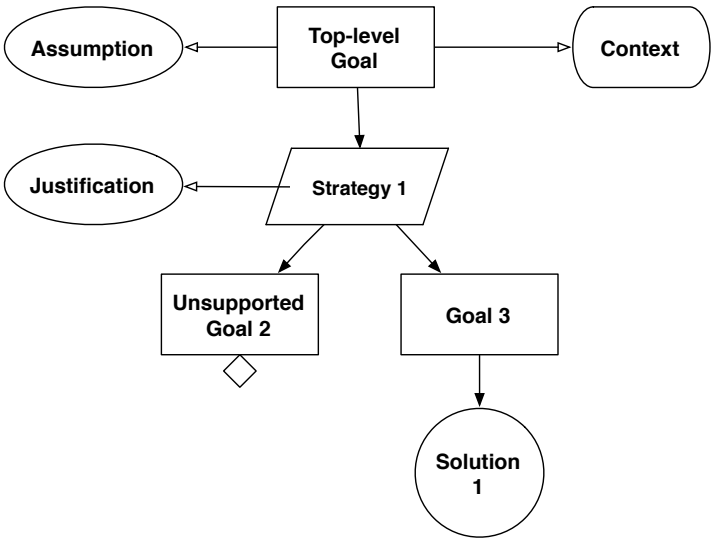


Figure 2.7: Assurance case with Goal Structure Notation

The GSN notation (Kelly, 1999; Kelly and Weaver, 2004) has more elements to express an assurance case. It has goal (claim), solution (evidence), strategies (argument),

context, assumption, and justification. Context provides information in which the goals are stated while Assumption indicates the assumptions made about the goal. Justification provides support for why a strategy is adopted. Figure 2.7 shows an example of an assurance case built with GSN notation. There is a notion of undeveloped goals, which are goals that have not yet been supported by an evidence or a solution.

Although assurance cases have been used in the safety community for demonstrating safety claims about systems (Bloomfield and Bishop, 2010; Kelly, 1999), they have also been used to support security claims. Weinstock et al. (2013) state that a security assurance case presents arguments, supported by evidence, of claims that systems exhibit certain security properties. Furthermore, assurance cases can be co-created with or during design of a system.

Chapter 3

A New Security Pattern Catalog

In this chapter, I report on a study of existing literature for security patterns using a strategy-based search. I have collected 279 security patterns into the catalog. In order to allow a uniform definition of security patterns, a pattern template has been defined. Many templates have previously been proposed using different fields to represent similar information. However, there is no standard template for security patterns. Therefore, I have proposed a new security pattern template, based on existing templates, to provide a uniform definition for security patterns.

In Section 3.1, I present the definition of the template and its fields. In Section 3.2, I describe our search strategy and lists the sources where I obtain the patterns. Finally, I present a portion of the security pattern catalog and metadata extracted from our catalog in Section 3.3.

3.1 Security Pattern Template

In order to allow a uniform definition of security patterns, a security pattern template needs to be defined. However, there is no standard security pattern template. Different authors have proposed their own templates. These templates vary in quality (Laverdiere

et al., 2006). A standard security pattern template is required and thus I define a new security pattern template that is based on several existing templates, and merges fields that are similar.

Existing pattern templates from three books on security patterns (Fernandez-Buglioni, 2013; Schumacher et al., 2006; Steel et al., 2005), one seminal work on security patterns (Yoder and Barcalow, 1997) and one book on design patterns (Gamma et al., 1995) are analyzed. The design patterns book is included as their template is the standard template in the design pattern community. There are 27 fields in total from these five templates. Some of the fields are named differently. For example "Alias" is used by Yoder and Barcalow (1997) to capture variant names for a pattern while Gamma et al. (1995) and Schumacher et al. (2006) call this "Also Known as". I adopt several fields, labelled (A), and merge (and/or rename) fields from existing templates, labelled (M). I propose a new field, **Source**, and label it (N). The fields for our security pattern template are: **Pattern Name** (A), **Intent** (M), **Problem** (M), **Solution** (M), **Alias** (M), **Participants** (M), **Security Properties** (A), **Interactions** (M), **Known Uses** (M), **Related Pattern** (M), and **Source** (N). The definition for each field is presented below. Table 3.1 shows how each of the included fields corresponds to fields in existing templates.

- **Pattern Name** *A pattern's name is a pattern identifier. It should be unique and meaningful. It should portray the essence of a pattern. This field is required as a pattern needs a unique identifier.*

- **Intent** *A short summary of the security problem(s) that the pattern is addressing and how it is solved. This field is included to allow for easy perusal of the pattern catalogue and understand the pattern quickly.*

- **Problem** *A recurring security problem that this pattern is addressing.*

- **Solution** *Solution to solve the problem that the pattern in addressing.*
- **Alias** *Different names that may be used for the pattern. This field is also used to capture variations of a pattern that are essentially the same but named differently.*
- **Participants** *The actors that are involved in the pattern. This field contributes to understanding the Solution.*
- **Interactions** *The interactions between actors. This field contributes to the Solution field.*
- **Security Properties** *The security properties that are affected, both positively and negatively, by the pattern.*
- **Known Uses** *This field provides examples of the pattern. It provides good evidence that this is a pattern, not just a once-off problem and solution.*
- **Related Pattern** *This fields shows the relationship of the pattern with other patterns. This includes other patterns that solve similar problems or that help to improve the pattern. This field also helps identify potential composition of patterns.*
- **Source** *This field identifies literature where a pattern has been introduced and described. A reference to the source enables others to look at the original text of a pattern and its broader context not captured in this template.*

The fields that are excluded are: example, context, example resolved, implementation, applicability, sample code, reality check, and security risks and factor. For each field, its definition and justification for exclusion from our pattern template are provided below. Table 3.2 shows the existing templates where these excluded fields appear.

Table 3.1: Existing templates where each of the included fields appears

My Field	Existing Field	Schumacher	Fernandez	Gamma	Yoder	Steel
Name	Name	x	x	x	x	x
Intent	Intent		x	x		
Problem	Problem	x	x		x	x
	Motivation			x	x	
	Forces				x	x
Solution	Solution	x	x		x	x
Participants	Structure	x	x	x		x
	Participants			x		
Interactions	Dynamics	x	x			
	Collaboration			x		
Security Properties	Consequences	x	x	x	x	x
Related Pattern	See Also	x	x			
	Related Pattern			x	x	x
Known Uses	Known Uses	x	x	x	x	
	Non-Security Known Uses				x	
Alias	Also Known As	x		x		
	Variant	x				
	Alias				x	
	Strategies					x
Source	-					

- **Example** *This field specifies a scenario where the problem that the pattern is addressing exists. Since the pattern has been used in existing system, the problem that the pattern is addressing exists. Therefore, this field is obviated by Known Uses field.*

- **Context** *This field suggests in which situation/condition the pattern is applica-*

ble. Situation in which a pattern can be applied varies and this field might not cover all the possible situations. A pattern might still be applicable even if this field does not cover a particular situation. Therefore, I leave out this field.

- **Example Resolved** *This field shows a scenario how the pattern solves the problem it is addressing. The Intent field includes the solution provided by the pattern and thus this field is omitted.*

- **Implementation** *This field elicits implementation hints and common implementation pitfalls. We dealt with implementation through design fragments and thus this field is dropped.*

- **Applicability** *see Context field. This field is the same as* **Context**.

- **Sample code** *This field provides code snippets to implement the pattern. This field is excluded as we dealt with implementation trhough design fragments. Furthermore, there are not many security pattern descriptions that provide sample code.*

- **Reality Check** *This fields contains information to demonstrate that the pattern is practical and feasible. This is covered by the* **Known Uses**.

- **Security Risks and Factors** *This field provides several considerations to be mindful of when applying/choosing the pattern. There is only one existing template that contains this field and thus I omit this field.*

3.2 Search Strategy

In this section, I discuss our search strategy for collecting literature that introduce and describe security patterns. I have not done a systematic literature review (Kitchenham,

Table 3.2: Existing templates where each of the excluded fields appears

Field	Schumacher	Fernandez	Gamma	Yoder	Steel
Example	x	x		x	
Context	x	x			
Example Resolved	x	x			
Implementation	x		x	x	
Applicability			x		
Sample Code			x		
Reality Check					x
Security Factor & Risks					x

2004) or a systematic mapping study (Petersen et al., 2008). I use a search strategy based on those methods.

First, the databases as sources to search for literature are identified. Then, the keywords to search for existing literature are defined. After which, the inclusion criteria to determine whether or not a particular literature is relevant are detailed. Finally, the criteria to filter out irrelevant literature are described.

The databases are used are:

- ACM Digital Library (DL) – `dl.acm.org`
- IEEE Xplore – `http://ieeexplore.ieee.org/Xplore/home.jsp`
- Science Direct – `http://www.sciencedirect.com/`
- Google Scholar – `https://scholar.google.com`

The databases listed above are used in the literature search. Science Direct and Google Scholar are included to expand the literature search space as they cover literature that are not published in either ACM or IEEE.

A list of relevant search strings are identified:

- (“Security” OR “design”) AND (“pattern” OR “patterns”)
- (“Secure” OR “security”) AND (“design pattern”)

In the first search string, “security” OR “design” is used to broaden the search to include either of these two terms. The term “pattern” OR “patterns” is used to include singular and plural forms of the term “pattern”. Then, the four terms are joined together using AND conjunction to query the databases. The second search string is included to search for literature that mentions either “secure design pattern” or “security design pattern”. The databases are queried using these search strings to search for relevant literature that are published between (inclusive) 1997 and 2014. 1997 is included because that is the year when the seminal paper on security patterns was published.

The inclusion criteria are:

- report studies in the security patterns context;
- propose solutions to address security problems and relate them to either security patterns or concepts similar to security patterns;
- adopt the concept of security pattern; or
- are written in English.

The exclusion criteria are:

- investigate implementation of existing security patterns but do not propose a security pattern;
- do not explicitly describe a security pattern; or
- only propose sample code implementation of a pattern without describing the pattern itself.

I have shortlisted 40 publications after filtering the search results based on the inclusion and exclusion criteria.

3.3 Security Pattern Catalog

In this section, I present a collection of security patterns, gathered from the literature survey. First is an illustrative example of our security pattern catalog. Then, the metadata of literature sources for these patterns is presented, including the publication type and number of patterns proposed, in Table 3.3, and distribution of security properties affected by the patterns in the catalog.

Table 3.3: Pattern Metadata. Most of these publications are conference papers, with the exception of some published as books (*) and technical reports (+)

Publication Title	**Reference**	**# patterns**
Password Patterns	Riehle et al. (2002)	16
A collection of privacy design pattern	Hafiz (2006)	9
Design Patterns for Fault Containment	Saridakis (2003)	3
Even more patterns for secure operating system	Fernandez et al. (2006)	3
privacy patterns for online interactions	Romanosky et al. (2006)	3
Privacy-aware network client pattern	Sadicoff et al. (2005)	1
Reverse Proxy Patterns	Sommerlad (2003)	3
A pattern language for firewalls	Fernandez et al. (2003)	2

Continued on next page

Table 3.3 – continued from previous page

Publication Title	Reference	# patterns
Securing the broker pattern	Morrison and Fernandez (2006*b*)	1
Security Design Patterns	Romanosky (2001)	8
Patterns for the extensible access control markup language	Delessy and Fernandez (2005)	3
A pattern language for security models	Fernandez and Pan (2001)	4
Architectural Patterns for enabling application security	Yoder and Barcalow (1997)	7
Credential Pattern	Morrison and Fernandez (2006*a*)	1
Firewall Patterns	Schumacher (2003)	3
More patterns for operating systems access control	Fernandez and Sinibaldi (2003)	4
Patterns for managing internet-technology systems	Dyson and Longshaw (2003)	5
Remote Authenticator/Authorizer	Fernandez and Warrier (2003)	1
A Pattern Language for designing and implementing role-based access control	Kodituwakku et al. (2001)	6

Continued on next page

Table 3.3 – continued from previous page

Publication Title	Reference	# patterns
Controlled Access Patterns	Elsinga and Hofman (2002)	2
Security Design Patterns+	Blakley and Heath (2004)	13
Security Taxonomy Pattern Language	Elsinga and Hofman (2003)	2
Security Patterns for Web Application Development+	Kienzle and Elder (2002)	29
The Authenticator Pattern	F. Lee Brown et al. (1999)	1
Object Filter and Access Control Framework	Hays et al. (2000)	1
Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*	Steel et al. (2005)	23
Security Patterns: Integrating Security and Systems Engineering*	Schumacher et al. (2006)	45
Two patterns for web services Security	Fernandez (2004)	2
Patterns for Access Control in Distributed Systems	Delessy et al. (2007)	3
Pattern Language for specification of communication protocols	Pärssinen and Turunen (2002)	17

Continued on next page

Table 3.3 – continued from previous page

Publication Title	Reference	# patterns
Credential Delegation: Towards Grid Security Patterns	Weiss (2006)	1
Security Patterns and Security Standards - Selected security patterns for anonymity and privacy	Schumacher (2002)	2
Tropyc: A pattern language for cryptographic software	Braga et al. (1998)	10
Session Patterns	Sørensen (2002)	7
A Pattern Language for Cryptographic Key Management	Lehtonen and Pärssinen (2001)	11
Patterns for operating system access control	Fernandez (2002)	5
Security Patterns for Agent Systems	Mouratidis et al. (2003)	4
Patterns for Application Firewalls	Delessy-Gassant et al. (2004)	2
Secure Design Patterns+	Dougherty et al. (2009)	15
	Total number of patterns	279

After analysing all the patterns in the catalog, 79 patterns were classified as not being design patterns. These are either recommendations (patch software frequently), procedural (such as how to generate good encryption keys securely) or naming schemes (e.g.

Alice and Friend by Lehtonen and Pärssinen (2001)). This leaves 200 security design patterns. These patterns were analyzed to identify which security properties are affected by each of the patterns. Figure 3.1 shows the distributions of security properties that are affected by the patterns in our catalog. The majority of the patterns in the catalog affect the confidentiality property (107), followed by integrity (90) and then availability (63). Note that a pattern can affect one or more properties. Furthermore, several patterns in the catalog do not affect confidentiality, integrity and availability but rather other properties such as maintainability and non-repudiation. These properties have been included as "Others".

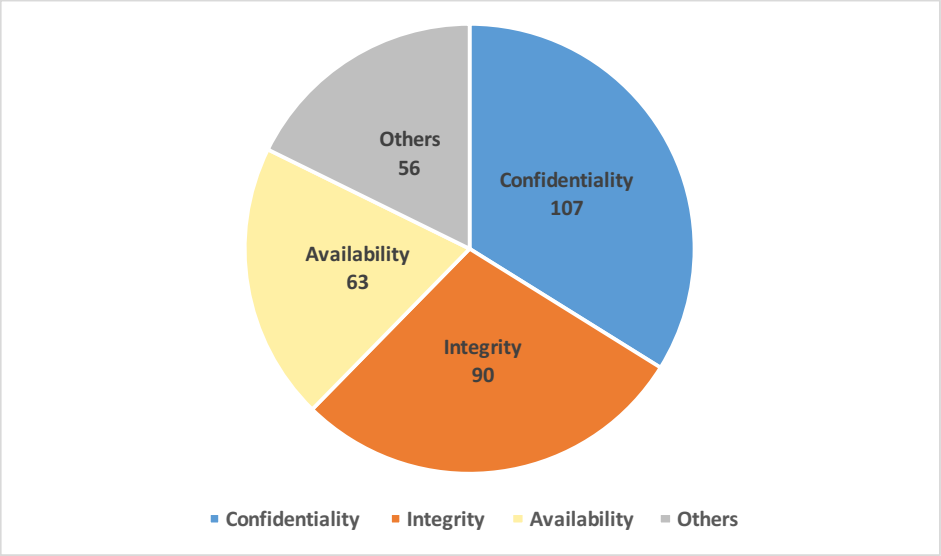


Figure 3.1: Security Properties Distribution

Below is an illustrative example security pattern from the catalog, using the template proposed in Section 3.1. Multiple prior descriptions of a pattern are captured in the *Alias* field. As an example, the *Authorization Enforcer* that is proposed by Steel et al. (2005) is similar to *Authorization* (Schumacher et al., 2006) and *Remote Authorizer* (Fernandez and Warrier, 2003).

- **Pattern Name** Authorization Enforcer

- **Intent** Defines the access policy for resources
- **Alias** Authorization (Schumacher et al., 2006), Remote Authorizer (Fernandez and Warrier, 2003)
- **Problem** Components need to verify that each request is properly authorized. A way to control access to resources is needed.
- **Solution** For each active entity, indicate which resources it can access and how.
- **Participants** Client, SecureBaseAction, Subject, PermissionCollection, AuthorizationProvider, AuthorizationEnforcer, AccessStore. Each of these can be viewed as a component
- **Interactions** The Client requests authorization from *SecureBaseAction* and sends *Subject*. *SecureBaseAction* uses the credential information in *Subject* and invokes *AuthorizationEnforcer*'s authorize method. *AuthorizationEnforcer* then requests the permissions of the client from *AuthorizationProvider*. *AuthorizationProvider* retrives permission from *AccessStore*, creates *PermissionCollection*, stores it into *Subject* and returns *Subject* to *AuthorizationEnforcer*. *AuthorizationEnforcer* then sends *Subject* back to *Client*.
- **Security Properties** Authorization, if done properly, promotes separation of responsibility through access rights. It defines which resources an entity can access and with what access rights. That positively affects confidentiality, integrity and availability.
- **Known Uses** It is used as the basis for access control in many products, such as Windows, UNIX, MySQL (Schumacher et al., 2006).
- **Related Pattern** Authentication Enforcer (Steel et al., 2005) is required to authenticate users.

- **Source** Steel et al. (2005)

Figure 3.2 shows the number of publications per year. The seminal work on security patterns was published in 1997 by Yoder and Barcalow. The publication number peaks in 2002 and 2003, with 9 and 10 publications respectively that describe/propose security patterns.

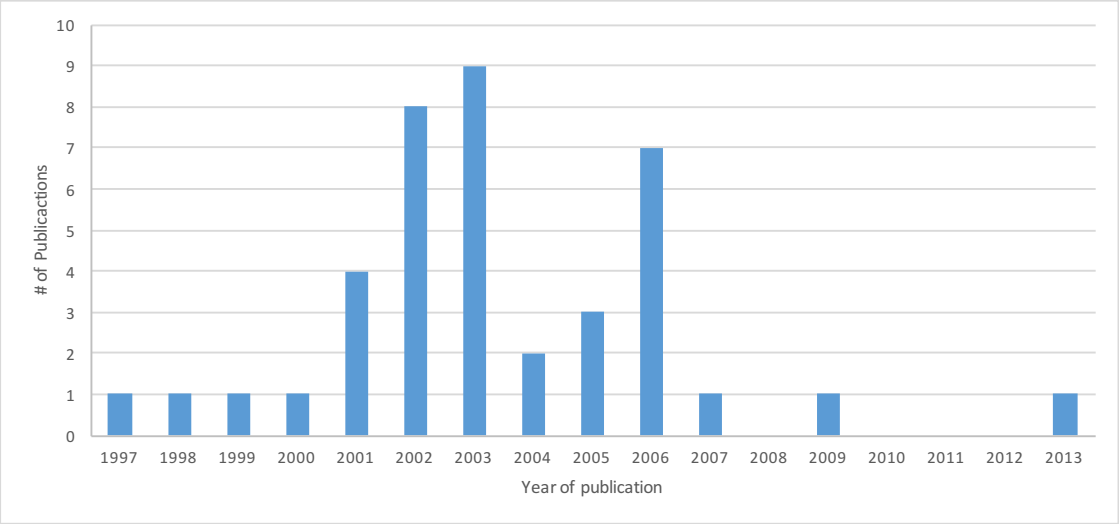


Figure 3.2: Pattern Publication Year Trends

Chapter 4

Capability-specific Design Fragments

In this chapter, I show how a security pattern can be formalized as a platform-specific design fragment, which allows for design-level verification. Although a security pattern provides a solution to a recurring security problem, its informal nature does not allow reasoning about its properties. A design fragment is a partial realization of a design pattern in the context of a particular platform. I describe the general concept of design fragments in Section 4.1.

In Section 4.2, I explain the concept of capability-specific design fragments: design fragments that target capability-based computing platforms. A capability-specific design fragment has two representations, i.e. graphical and textual, which are described in Section 4.3. I then describe how capability-specific design fragments can be derived from security patterns in Section 4.4. In Section 4.5, I provide examples of capability-specific design fragments derived from some of the security patterns from the catalog presented in Chapter 3.

4.1 What is a Design Fragment?

Design fragments are proposed in this thesis as a partial realization of a design pattern in the context of a particular platform. They specialize the *solution* field of a pattern to describe how the mechanisms in that platform are used to address the intent of the pattern. Design fragments aim to be reusable within this context. A design fragment can be represented either using a drawing of components and connections, or using a formal specification language. A pattern is generally informal and does not allow for reasoning about its properties. A formal specification of a design fragment allows for verification to provide assurance that it achieves its intended goal, including correctness or security properties.

The quality of patterns' documentation varies (Laverdiere et al., 2006). Insufficient documentation can lead to many interpretations, which may lead to design flaws and affect the assurance that is provided for the patterns (Heyman et al., 2012). Providing assurance about the properties of a system is essential, especially systems that are security-critical. As patterns are generally informal, formal reasoning about their properties to provide assurance is infeasible. Therefore, I introduce the concept of a design fragment to provide assurance about patterns through reasoning about its properties, within the constraint of a particular platform.

Another advantage for having a design fragment is that it incorporates the mechanisms provided by the underlying platform. This reduces the required step of manually translating the design to a specific platform for implementation purposes.

In many pattern templates (Fernandez-Buglioni, 2013; Gamma et al., 1995; Schumacher et al., 2006; Yoder and Barcalow, 1997), the Known Uses field is included to capture examples of the pattern being used in existing systems, sometimes in different domains. Different systems may use different platforms. However, the content of the Known Uses field is often very high level. For instance, the Known Uses field of

the authenticator security pattern (Fernandez-Buglioni, 2013, p. 56) contains (quoting the authors): "e-commerce sites, such as eBay and Amazon," and "Commercial operating systems use some form of authentication, typically passwords, to authenticate their users". In the design patterns book (Gamma et al., 1995, p. 329-330), the Known Uses field of the Template Method pattern says that "template methods are so fundamental that they can be found in almost every abstract class.".

4.2 Capability-specific Design Fragment

A capability-specific design fragment is a specialization of a design pattern for capability-based platforms. A capability-specific design fragment takes into consideration the security mechanisms provided by the underlying capability-based platform, especially the access rights of the platform.

As discussed in the literature review in Chapter 2, there are similarities between the different capability concepts, even though they are implemented differently. The similarities are that: 1) capabilities are transferable and are one-directional; and 2) access rights, represented by capabilities, are corroborated by the principals/actors. Different platforms might have different levels of granularity for access rights. A pure capability-based platform, such as EROS (Shapiro et al., 1999) and KeyKOS (Hardy, 1985), has only one access right, where having a capability means having all the access rights (i.e. read, write, grant). In contrast, seL4 (Klein et al., 2009), which implements the object-capability concept, has four different access rights: read, write, grant and mint. This difference affects how patterns will be instantiated.

In my proposed approach for incrementally building and verifying an application design, I instantiate security patterns for specific capability-based platforms. Each security pattern has security goal(s) that its instantiation needs to satisfy. Capability-based

design fragments allow for verification to provide assurance that the patterns' goals are met. This verification is done using techniques referred to as a verification procedure. Verification procedures are intended to be reusable and will be discussed in Chapter 6.

As pointed out by Miller et al. (2003), the vast majority of implemented capability-based systems implements the object-capability model. Serscis Access Modeller (SAM) (Leonard et al., 2013) is a modeling tool for object-capability systems and thus I have used SAM as a modeling tool for capability-based design fragments in this thesis.

4.3 Design Fragment Representations

There are two representations of the capability-specific design fragments: textual and graphical. The textual representation consists of component specifications, initial capability distributions for component, component initialization, initial triggers and the intended security goal.

Component specifications consist of the component structure, exported functions, and the behavior of those functions. Each function has a return value, function name, function parameters and function body. The key characteristic of the behavior that I am interested in is the invocation of other components and the capability propagation through those invocations.

The security goals of the security pattern are captured using the security property template defined in Section 6.3. This is then checked by defined verification procedures, which are used to verify that the design fragment achieves its intended goals.

A graphical representation of the design fragment can also be generated. This shows the capability distribution among components of the design fragments and shows a state of the design fragment where all the possible capability propagations are executed. The transitive closure of all capability propagations is then checked against the verification

procedure to verify whether the design fragment achieves its security goal.

In SAM, the component specification is written in a Java-like language. There are three return types that a function can have, which are *Void*, *Value* and *Ref*. *Void* means the function returns normally but does not supply a value. *Value* is a special type that encompasses every data structures, such as string, integer. *Ref* means the function returns a capability to a particular component that the component has access to. Function parameters can either be *Value* or *Ref* types. The function body contains the implementation detail of that particular function. In the function body, the focus is on specifying the invocation of other components. The implementation details of the function are left to the developer to complete.

The security goal of a security pattern is represented using Datalog rules (Ceri et al., 1989). The Datalog rules are the procedures in the verification procedure the verification procedures of the design fragment, which will be detailed in Chapter 6. Datalog is an example of formal representations of the security goals.

The graphical representation of the design fragment is generated from its textual representation in SAM. It is represented as a graph whose nodes represent components and whose directed edges represent capabilities (with an arrow pointing from one component to another if the source component holds a capability to the destination component).

4.4 Deriving Design Fragments from Security Patterns

I develop design fragments by extracting important information from security patterns. The extracted information includes the goal of the pattern, the actors involved, the main functionalities of each actor, the interaction between different actors in the pattern, and the implicit assumptions of the pattern.

First, the actors that are involved in a pattern are determined by going through the

text manually, focusing on nouns that perform at least one action. Each actor is modeled as a component in the design fragment. The actors are then classified into one of four different types: *initiator*, *member*, *controller*, and *sub-controller*. *Initiator* is a type of component that triggers the flow of a pattern. *Member* is a type of component that provides specific functionalities. *Controller* is a type of component that coordinates the flow of a pattern and invokes the functionalities of its members. It is also the entry point of a pattern. A *sub-controller* is a type of component that coordinates the flow of the pattern, by invoking the functionalities of its members. A sub-controller must be a member of a controller.

Once the actors have been identified, their interactions are identified. This is done in a review of the pattern's text. The interactions determine the initial capability distribution for the design fragment. After which, the actors classification is checked to ensure that the actors are classified correctly and the actors are reclassified to rectify any errors.

Identifying interactions also helps to establish the main functionalities of each actor. The functionalities of each actor are mapped on to the behaviors of the corresponding component.

All this information is utilized to manually create the capability-specific design fragments. Each actor and their corresponding functionality is represented as a component type and corresponding behavior. The interactions between the actors also helps to identify the initial capability distribution of the design fragment. An initial capability distribution is an initial state of the design fragment where each component is granted necessary capabilities.

The goal of the pattern is extracted following a template defined in Section 6.3 and translate it into a verification procedure. This verification procedure is used to verify whether the design fragment satisfies its intended security properties.

4.5 Examples

In this section, I present two examples of realizing security patterns into capability-specific design fragments.

4.5.1 Secure Logger Pattern

The secure logger security pattern (Steel et al., 2005) has two goals:

- decouple logging functionality for maintainability; and
- ensure that the contents of log file remains confidential from unauthorized access.

The first goal is a non security-related goal and is aimed at improving the maintainability of a system. The second goal is a security-related goal, which concerns confidentiality.

There are several actors in this pattern: *client*, *secureLogger*, *logManager*, *logFactory* and *logger*. Each actor is modeled as a component class in the design fragment.

The interactions between different actors define what functions each component class will need to export and invoke. Furthermore, these interactions also inform the initial capability distribution required for the baseline model to work. The baseline model is one in which every component behaves as specified, i.e. all components are trusted. The constructor for each component class is defined based on the initial capability distribution.

The interactions between the actors in the secure logger pattern are as follows: *Client* sends a log command to *secureLogger*, together with the data to be logged. Upon receipt, the *secureLogger*, whose main responsibility is to collect the data, sends the data with a log command to the *logManager*. The *logManager* will request a new instance of *logger* from *logFactory*. The *logger* is the component that logs data. It creates a new *file* and

writes data into that file. *Client* is classified as an initiator, *secureLogger* is classified as a controller, *logManager* is classified as a sub-controller while *logFactory*, *logger* and *file* are classified as members.

Table 4.1 shows the functions that different components need to implement and invoke and the capabilities required.

Table 4.1: Secure Logger Design Fragment Actors and Interactions

Component	**Role**	**Fn Required (Params)**	**Fn Return Type**	**Components & (Fns) Invoked**	**Capabilities**
client	initiator	log (Value:data)	Value	*secureLogger*: (logMsg)	*secureLogger*
secureLogger	controller	logMsg (Value:data)	Value	*logManager*: (logMsg)	*logManager*
logManager	sub-controller	logMsg (Value:data)	Value	*logFactory*: (newLogger) *logger*: (write)	*logFactory*
logFactory	member	newLogger (void)	Ref	*logger*: (constructor)	-
logger	member	write (Ref:file, Value:data)	Value	*file*: (write)	-
file	member	write (Value:data)	Value	-	-

The client has a capability to the *secureLogger*. The *secureLogger* has a capability to the *logManager*. The *logManager* has a capability to both the *logFactory* and the newly created *logger*.

The textual representation of the Secure Logger design fragment is as shown in Listing 4.1. The first part of the textual representation shows each component's specification. The code under `config` declares the necessary components while the code under `setup` shows the initial capability distribution, e.g. `logManager` has an access to

`logFactory` and `file`. Finally, the code under `test` triggers the flow of the pattern, with `client` as the initiating component.

Listing 4.1: Secure Logger Design Fragment

```
1  class Client{
2    private Ref secureLogger;
3    public Client(Ref sl){
4       secureLogger = sl;
5    }
6    public Value log(Value data){
7       return secureLogger.logMsg(data);
8    }
9  }
10 class SecureLogger{
11   private Ref logManager;
12   public SecureLogger(Ref lm){
13     logManager = lm;
14   }
15   public Value logMsg(Value data){
16     return logManager.logMsg(data);
17   }
18 }
19 class LogManager{
20   private Ref logFactory;
21   private Ref file
22   public LogManager(Ref lf, Ref f){
23     logFactory = lf;
24     file = f;
```

```
25    }
26    public Value logMsg(Value data){
27      Ref logger = new logFactory.newLogger();
28      return logger.write(file,data);
29    }
30  }
31  class LogFactory{
32    public LogFactory(){}
33    public Ref newLogger(){
34      Ref logger = new Logger();
35      return logger;
36    }
37  }
38  class Logger{
39    public Logger(){}
40    public Value write(Ref file, Value data){
41      return file.write(data);
42    }
43  }
44  class File{
45    public File(){}
46    public Value write(Value data){
47      //write to file
48      return "success";
49    }
50  }
51  config{
52    SecureLogger secureLogger;
```

```
53    LogManager logManager;
54    logFactory logFactory
55    File file;
56    Client client;
57    setup{
58      logFactory = new LogFactory();
59      file = new File();
60      logManager = new LogManager(logFactory,file);
61      secureLogger = new SecureLogger(logManager);
62    }
63    test{
64      client = new Client(secureLogger);
65      Value data;
66      client.log(data);
67    }
68  }
```

We can verify that the user does not have direct access to the file. A Datalog rule for the Points-To analysis engine in SAM can analyze this fragment to check that the user does not have access to the file, using the query `!hasRef(<user>,<file>)`.

Figure 4.1 shows the graphical representation of the design fragment that is generated by SAM. It shows the state where the *secureLogger* has a capability to *logManager*. The *logManager* has a capability to both the *logFactory* and the *file*. The *logManager* then gained a capability to the *logger*, which is created during execution time.

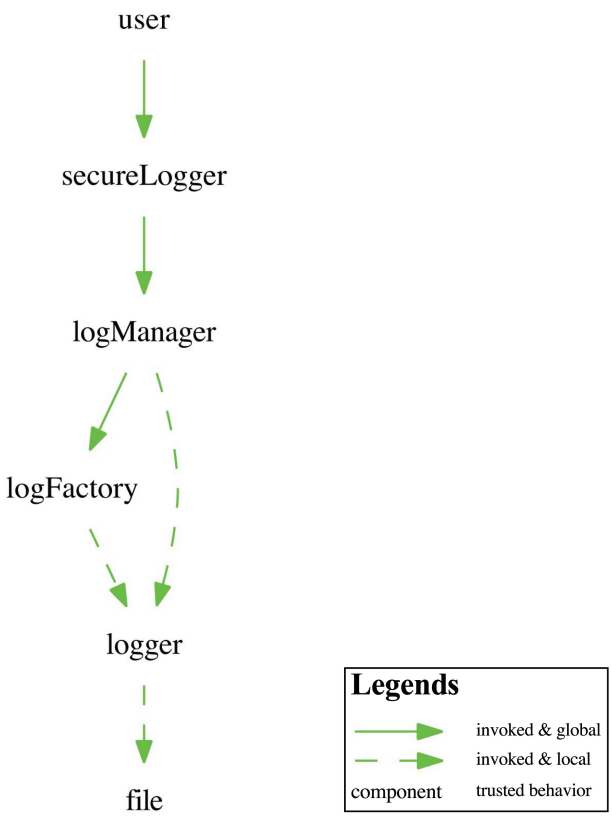


Figure 4.1: Secure Logger Graphical Representation (baseline)

4.5.2 Encrypted Storage Pattern

The encrypted storage pattern (Kienzle and Elder, 2002) aims to harden the confidentiality of a system. It encrypts data before storing it, and the encryption key must be stored securely. This mitigates the impact of the loss of a file to an attacker, because the content of the file remains confidential, as it has been encrypted.

There are several actors in this pattern, namely the *client*, *encryptedStorage*, *storage*, *encryptDecrypt* and *key*. Each actor is modeled as a component class in the design fragment. The interactions between different actors define what functions each component class will need to export.

The interactions between the actors in the encrypted storage pattern are as follows:

Client sends a store command to *encryptedStorage*, together with the data to be stored. Upon receipt, the *encryptedStorage*, whose main responsibilities are to collect the data and to orchestrate the appropriate process, sends the data to *encryptDecrypt* with an encrypt command. *encryptDecrypt* then encrypts the data and return the encrypted data to *encryptedStorage*. *encryptedStorage* then sends the encrypted data to *storage* with a store command. *storage* then store the data. During the initial setup, *encryptedStorage* loads the value of the key to *encryptDecrypt* and keeps to itself the capability to the key.

Table 4.2: Encrypted Storage Design Fragment Actors and Interactions

Component	**Role**	**Fn Required (Params)**	**Fn Return Type**	**Components & (Fns) Invoked**	**Capabilities**
client	initiator	write (Value:data)	Value	*encryptedStorage* (write)	*encrypted-Storage*
encrypted-Storage	controller	write (Value:data)	Value	*storage* (write) *encryptDecrypt* (encrypt) *key* (getValue)	*storage* *encrypt-Decrypt* *key*
storage	member	write (Value:data) read (Value:data)	void Value	-	-
encrypt-Decrypt	member	encrypt (Value:data) decrypt (Value:data)	Value Value	-	-
key	member	getValue (void)	Value	-	-

Table 4.2 shows the role that each component assumes and the functions that each components needs to implement. It also shows different functions that each component

invoke and the capabilities required.

The client has a capability to *encryptedStorage*. *encryptedStorage* has a capability to *storage*, *encryptDecrypt* and *key*. In this design fragment, the *client* assumes the role of an initiator and *encryptedStorage* assumes the role of a controller. *storage*, *encryptDecrypt* and *key* are members.

The textual representation of the encrypted storage design fragment is shown in Listing 4.2. The first part of the textual representation shows each component's specification. The code under `config` declares the necessary components while the code under `setup` shows the initial capability distribution, e.g. the `client` has an access to the `encryptedStorage`. Finally, the block of code under `test` triggers the flow of the pattern, with the `client` as the initiating component invoking store command of `encryptedStorage`.

Listing 4.2: Encrypted Storage Design Fragment

```
1  class File{
2     public void write(Ref data){}
3     public void read(){}
4  }
5  class Key{
6    Value myValue;
7    public Value getKey(){
8      return myValue;
9    }
10 }
11 class EncryptDecrypt{
12   private Ref myKey;
13   public EncryptDecrypt(){}
14   public void loadKey(Value key){
```

```
15      myKey = key;
16    }
17    public Value encrypt(Value data){
18      /* do encryption */
19      return data;
20    }
21    public Value decrypt(Value data){
22      /* do decryption */
23      return data;
24    }
25  }
26  class Storage{
27    public void store(Ref data){}
28    public Ref retrieve(Ref data){
29      /* retrieve data (encrypted) */
30      return data;
31    }
32  }
33  class EncryptedStorage{
34    private Ref myKey;
35    private Ref myEncryptDecrypt;
36    private Ref myStorage;
37    private Value keyVal;
38    public EncryptedStorage(Ref key, Ref ed, Ref storage){
39      myKey = key;
40      myEncryptDecrypt = ed;
41      myStorage = storage;
42    }
```

```
43    public void loadKey(){
44      keyVal = myKey.getKey();
45      myEncryptDecrypt.loadKey(keyVal);
46    }
47    public void storeData(Ref data){
48      Ref encryptedData = myEncryptDecrypt.encrypt(data);
49      myStorage.store(encryptedData);
50    }
51    public Object retrieveData(Ref Client){
52      Ref rawData = myStorage.retrieve(Client);
53      Ref data = myEncryptDecrypt.decrypt(rawData);
54      return rawData;
55    }
56  }
57  class Client{
58    private Ref server;
59    public Client(Ref serverAdd){
60      server = serverAdd;
61    }
62    public void store(Ref data){
63      server.storeData(data);
64    }
65    public void get(){
66      server.retrieveData();
67    }
68  }
69  config{
70    EncryptDecrypt encryptDecrypt;
```

```
71    Key key;
72    Storage storage;
73    EncryptedStorage encryptedStorage;
74    Client user;
75    setup{
76      storage = new Storage();
77      encryptDecrypt = new EncryptDecrypt();
78      key = new Key();
79      encryptedStorage = new EncryptedStorage(key,encryptDec-
80                                                  rypt,storage);
81    }
82    test{
83      Ref data;
84      user = new Client(encryptedStorage);
85      encryptedStorage.loadKey();
86      user.store(data);
87    }
88  }
```

We can verify that all the components, except *encryptedStorage*, do not have access to the *key*. A Datalog rule for the Points-To analysis engine in SAM can analyze this fragment. The listing below ensures that there is no component, except *encryptedStorage*, that has access to the *key*.

Figure 4.2 shows the graphical representation of the design fragment that is generated by SAM. It shows the state where *user* has a capability to *encryptedStorage*. The *encryptedStorage* has a capability to *storage*, *encryptDecrypt*, and *key*.

Listing 4.1: Verification Procedure of the Encrypted Storage Design Fragment

```
1 keyBreached(?Src,?T) :-
2     !MATCH(?Src, ?T),
3     !MATCH(?Src,<encryptedStorage>),
4     hasRef(?Src,?T).
5 assert !keyBreached(?Src,<key>).
```

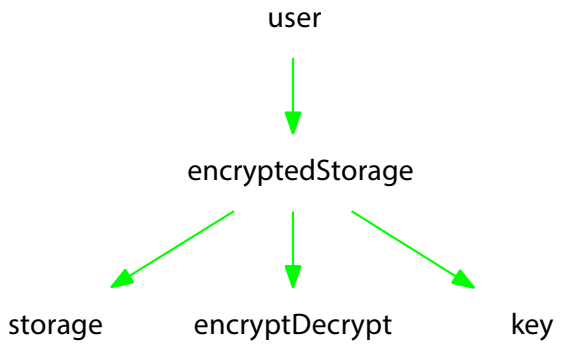


Figure 4.2: Encrypted Storage Design Fragment

Chapter 5

Composition of Design Fragments

In this chapter, I show how to compose design fragments together to build a secure application design. A design fragment can be composed with another design fragment or with an existing design. I achieve this by utilizing the primitive tactics that are defined in this chapter. Each of these primitives is then proven by induction to preserve a general security property that is defined in this chapter, the *Protected By* property.

Designing a secure system may require the composition of several security patterns, each concerned with different security properties. The first step is selecting the appropriate security patterns, i.e. those that support the security requirements of the application and that help mitigate attacks. These patterns are then specialized into design fragments as described in Chapter 4, which can then be composed with each other or composed with an existing application design. In both cases, I call this *composition*. The challenge in composing design fragments is producing a design that provides the intended security properties and does not break any security properties already present in the application.

There are two things I reuse when composing design fragments. First, I reuse the structure and behavior of the design fragment. Second, I reuse the verification procedure that I applied to the design fragment including both the query and Datalog rules defined

for the design fragments (see Chapter 6).

Each system is made up of components and connections. A connection represents a capability. It has two attributes, namely the source component and target component. A connection signifies that the source component has a capability over the target component.

In order to define this precisely, I make the following definitions.

Definition 1 *A* system *is* $\Sigma = \langle C, N \rangle$*, where C is a component set* $\{c_1, c_2, ..., c_n\}$*, and N is a connection set,* $\{n_1, n_2, ..., n_m\}$*, where each* $n_j = \langle c_i, c_k \rangle$ *for some* $c_i, c_k \in C$*.*

Each component set is divided into three disjoint subsets, namely S, T and all the rest. S is a set of components which are or hold the secrets or attributes that need to be protected in the system. T refers to a set of components whose behavior are trusted, i.e. it will only behave as specified. The specified behavior of a trusted component should be verified not to distribute capabilities to a secret. Each of the trusted components needs to be verified to ensure that it behaves as specified. Each secret, $s \in S$, is associated with its own set of components that are allowed access to it, G_s, which is referred to as a secret's granted set. For a component to be allowed access to a secret, that component needs to be added into the granted set of that secret. Granted components can be though of as internal to the system — protected operationally from direct access by external attack. A component with trusted behavior blocks further access propagation of a secret. The remaining components are neither secret nor a component with trusted behavior. I refer to systems with these two subsets defined, and with the property that only components in a secret's granted set, G_s, can have access to it, as *acceptable systems*. The property is called *Protected By*.

Definition 2 *A system of interest* $\Sigma_{int} = \langle C, N, T, S, G \rangle$ *extends a system* $\Sigma = \langle C, N \rangle$*, where T is a set of components that have trusted behavior, S is a set of components that*

need to be protected (secrets) and G is a indexed set of components, where $\forall\, s \in S, G_s \subset C$ *s.t.* G_s *is a set of components allowed to have access to s.*

Definition 3 *ProtectedBy(*Σ_{int}*), is defined by* $\langle x, c\rangle \notin (N \rhd T)^*$, *where* $c \in S \wedge x \notin G_c \wedge c \in C$

Definition 4 *An acceptable system is an interesting system that has the Protected By property.*

The *Protected By* property defines systems where only components that are in a secret's granted set can have direct access to it. In Definition 3, I consider all the possible access propagations in a system that can be performed by all the untrusted components. To define this, I first remove all the connections that point to a trusted component from the set of connections (N) by using range subtraction operation, $N \rhd T$. Then, I take the transitive closure of that set, $(N \rhd T)^*$, to propagate all the possible accesses. Finally, I check that for each secret, there is no connection to it from a component that is not in its granted set.

5.1 Composition Tactics

I have identified six primitives to compose design fragments, which I call composition tactics. These tactics are connect, disconnect, create, delete, grant, and revoke. Each tactic affects either the component and/or the connection set and thus affects the verification procedures of the composite system. As mentioned in Chapter 4, each system (including design fragments) is associated with verification procedures. These procedures are Datalog rules and queries that are used to verify that a system has certain intended security properties, which are specified using the template that will be defined in Chapter 6. Based on this template, components that are not in the granted set cannot have access to a secret. To be more precise, each verification procedure acts on the

component sets. I prove the soundness of the tactics in Section 5.2. I derive higher-level tactics using these primitives and show several examples in Section 5.3. Finally, I discuss how composition of design fragments support a security assurance case in Section 5.4.

5.1.1 Connect Tactic

The connect tactic combines two systems together to form a larger system. This is done by creating an edge between two nodes in the connection graph. Structurally, the source component is granted a capability to the target component and invokes a particular function of the target component. I define connect in Definition 5.

The connect tactic requires two parameters: the source component and the target component.

Definition 5 *Connects to*

If $\Sigma^a = \langle C^a, N^a, T^a, S^a, G^a \rangle$ *and* $\Sigma^b = \langle C^b, N^b, T^b, S^b, G^b \rangle$ *then connecting* Σ^a *and* Σ^b *through* c_a *and* c_b *means creating* $\Sigma^c = \langle C^a \cup C^b, N^a \cup N^b \cup \{\langle c_a, c_b \rangle\}, T^a \cup T^b, S^a \cup S^b, G^a \cup G^b \rangle$, *where* $c_a \in C^a$ *and* $c_b \in C^b$ *and* $\langle c_a, c_b \rangle \notin N^a \cup N^b$ *with the restrictions that:*

> *For all secret nodes, d, where* $d \in S$ *and* $(c_b = d)$ *or* $(c_b \in (G^a \cup G^b)_d)$, *then:*
> $c_a \in (G^a \cup G^b)_d \wedge ((c_a \in (T^a \cup T^b)) \vee (\forall\, c. \langle c, c_a \rangle \in N \rightarrow c \in (T^a \cup T^b) \wedge c \in (G^a \cup G^b)_d))$

There are two cases where the connect tactic are restricted: 1) the target component is a secret; and 2) the target component is a component that is granted access to a secret. In the first case, the source component needs to be in the granted set of the target component (i.e. a secret). In the second case, where the target component is in a granted set of a secret, the source component needs to be allowed access to that secret as well.

Furthermore, the source component also needs to either have trusted behavior or each of the components connected to the source component needs have trusted behavior and is allowed access to that secret.

The restriction that is placed on the connect tactic is that the source component needs to be in the granted set of a secret

The connect tactic affects a verification procedure in two different ways, depending on the effect of the connect tactic on the structure of the system. First, if the connect tactic joins two systems together, it increases the total number of components in the resulting system (i.e. the sum of the number of components in each system). This increases the number of checks that need to be performed for each verification procedures in the resulting system. Second, if the connect tactic is performed between two components in the same system, it does not affect the number of components in the system. It increases the number of connections, including transitive connections, in the system.

5.1.2 Disconnect Tactic

The disconnect tactic removes a connection between a source component and a target component. This is done by removing a connection between two nodes in the connection graph. Structurally, a capability to the target component is revoked from the source component. (It may be that this gives us two completely disconnected systems.)

The disconnect tactic requires two parameters, i.e. the source component and the target component.

Definition 6 *Disconnects from*

If $\Sigma = \langle C, N, T, S, G \rangle$*, then disconnecting* $c_a \in C$ *from* $c_b \in C$ *means deleting a connection,* $\langle c_a, c_b \rangle \in N$*, and creating* $\Sigma' = \langle C, N', S, G \rangle$*, where* $N' = N \setminus \langle c_a, c_b \rangle$

5.1.3 Create Tactic

The create tactic creates a new component in the system. This is done by creating a new node in the connection graph. Structurally, a new component is initialized without holding any capabilities.

The create tactic requires two parameters, i.e the component name and the component type. Component type can be either secret, trusted or non-trusted component. Creating a secret requires an additional piece of information: a set of components that are allowed to have access to that secret.

Definition 7 *Creates*

If $\Sigma = \langle C, N, T, S, G \rangle$*, then creating a new non-secret, non-trusted component,* $c \notin C \wedge c \notin S \wedge c \notin T$*, means creating* $\Sigma' = \langle C \cup \{c\}, N, T, S, G \rangle$*. Creating a new non-secret trusted component,* $c \notin C \wedge c \notin S$*, means creating* $\Sigma' = \langle C \cup \{c\}, N, T \cup \{c\}, N, S, G \rangle$*. Creating a new secret component,* $s \notin C \wedge s \notin S$*, means creating* $\Sigma' = \langle C \cup \{s\}, N, T, S \cup \{s\}, G' \rangle$*, where* $G' = G \cup \{s \mapsto \{$*granted components to* $s\}\}$

Definition 7 covers the case for creating the first component for a system. When creating the first component, regardless of the type, of a system, $\Sigma = \langle \{\}, \{\}, \{\}, \{\} \rangle$. After the creation of that first component (c, the system will then be $\Sigma' = \langle \{c\}, N, T, S, G \rangle$, where c may be in one of S or T or neither.

As the create tactic increases the number of components in the system, it increases the number of checks that need to be performed in a verification procedure (or each of the existing verification procedures if there are more than one verification procedure).

5.1.4 Delete Tactic

The delete tactic removes a component in the system. This is done by deleting a node in the connection graph. Structurally, the target component is removed from the system

at design time. However, the component needs to be isolated before the delete tactic is allowed. A component is isolated when it has no capability with respect to other components in the system and there is no capability pointing towards that component. One way to isolate a component is to remove all the capabilities that the target component has and also remove any capabilities that points towards this component.

The delete tactic only requires one parameter, i.e. the component to be removed.

Definition 8 *Deletes*

If $\Sigma = \langle C, N, T, S, G \rangle$*, then deleting a non-secret, non-trusted component,* $c \in C \wedge c \notin S \wedge c \notin T$*, means creating* $\Sigma' = \langle C \setminus \{c\}, N, T, S, G \rangle$*. Deleting a non-secret, trusted component,* $t \in T \wedge t \notin S$*, means creating* $\Sigma' = \langle C \setminus \{t\}, N, T \setminus \{t\}, S, G \rangle$*. Deleting a secret component,* $s \in S$*, means creating* $\Sigma' = \langle C \setminus \{s\}, N, T, S \setminus \{s\}, G' \rangle$*, where* $G' = G \setminus \{s \mapsto \{$*granted components to* $s\}\}$ *Delete is allowed iff the component is isolated, i.e.* $\forall c_i \in C \,.\, \langle c, c_i \rangle \notin N \wedge \forall c_i \in C \,.\, \langle c_i, c \rangle \notin N$

As the delete tactic decreases the number of components in the system, it reduces the number of checks that need to be performed in a verification procedure (or each of the existing verification procedures if there are more than one verification procedure). Furthermore, I check whether the component that is to be deleted, is part of the granted set of a secret, s. If it is, I remove the (to-be-deleted) component from the secret's granted set.

5.1.5 Grant Tactic

The grant tactic allows a component to have access to a secret. This is done by adding a component to the set of components that are allowed to have access to a secret. This tactic requires two parameters: source component and target component. The restriction is that the target component is a secret component. I define grant in Definition 9.

Definition 9 *Add to Granted set*

If $\Sigma = \langle C, N, T, S, G \rangle$*, then allowing (granting)* c_i *access to* c_j *means creating* $\Sigma' = \langle C, N, T, S, G' \rangle$*, where* $G' = G \oplus \{c_j \mapsto G_{c_j} \cup \{c_i\}\}$ *iff* $c_j \in S$*. Otherwise,* $G' = G$*.*

The grant tactic loosens the restriction of the connect tactic. It affects the verification procedure by excluding more components to be checked for a particular secret.

5.1.6 Revoke Tactic

The revoke tactic disallows a component to have access to a secret. This is done by removing a component from the set components that are allowed to have access to a secret. This tactic requires two parameters: source component and target component. There are two restrictions for this tactic: 1) the target component is a secret component; and 2) there is no connection from source component to target component. I define revoke in Definition 10.

Definition 10 *Revoke from Granted set*

If $\Sigma = \langle C, N, T, S, G \rangle$*, then revoking* c_i *access to* c_j *means creating* $\Sigma' = \langle C, N, T, S, G' \rangle$*, where* $G' = G \oplus \{c_j \mapsto \{G_{c_j} \setminus \{c_i\}\}$ *with the restrictions:*

- $c_j \in S \wedge c_i \in G_{c_j} \wedge \langle c_i, c_j \rangle \notin N$
- *if* $(c_i \notin T \wedge c_i \notin S)$ *then* $\nexists \langle c_i, c \rangle \in N \cdot c \in G_{c_j} \wedge c \notin T \wedge c \notin S$

The revoke tactic tightens the restriction of the connect tactic. It affects the verification procedure by removing more components to be checked for a particular secret.

5.2 Composition Tactic Soundness

I prove that any sequence of the primitive tactics preserves the *Protected By* property by inducting over this list of primitives. At each step, I show that the tactics will always

preserve the *Protected By* property. When all interactions with a secret are mediated by its trusted set of components, I say that the secret is *Protected By* its trusted set. I provide the following definition to be more precise.

The structural induction performed will give assurance that, given each tactic is property preserving, any combination of the tactics will always preserve the *Protected By* property.

5.2.1 Connect Tactic

The connect tactic preserves the *Protected By* property.

Theorem 1 *Connect tactic will always preserve the Protected By property*

PROOF. Let $\Sigma^a = \langle C^a, N^a, S^a, G^a \rangle$ and $\Sigma^b = \langle C^b, N^b, S^b, G^b \rangle$, and Σ^a, Σ^b have the *Protected By* property by the inductive hypothesis.
Assume there are two components, c_a and c_b, where $c_a \in C^a$ and $c_b \in C^b$.
Connecting c_a to c_b means

$$\Sigma' = \langle C^a \cup C^b, N^a \cup N^b \cup \{\langle c_a, c_b \rangle\}, S^a \cup S^b, G^a \cup G^b \rangle \tag{5.1}$$

In order to prove Theorem 1, I need to identify all the possible cases where connect tactic can be applied and to prove that connect preserves the property in each of these cases. A connection has two parameters, i.e. source component (c_a) and target component (c_b). For the components, c_a and c_b, there are three possible cases where connect tactic can be applied:

- c_b is not a secret. i.e. $c_b \notin S^a \cup S^b$
- c_b is a secret and c_a is granted access to c_b. i.e. $(c_b \in S^a \cup S^b) \wedge (c_a \in (G^a \cup G^b)_{c_b})$
- c_b is a secret and c_a is not granted access to c_b. i.e. $(c_b \in S^a \cup S^b) \wedge (c_a \notin G^a \cup G^b)$

The effect of the connect tactic for the first two cases is as shown in Function definition 5.1. In these two cases, the resulting system has the property as there is no

connection to any secrets from a component that is not in a secret's granted set. In the last case, *connect* tactic is not allowed due to the restriction for the definition of *connect*, Definition 5. This restriction is that only component in the granted set of a secret can make a connection to that particular secret. As I start with systems, Σ^a and Σ^b, that have the *Protected By* property, the resulting system, Σ', after applying the connect tactic maintains its *Protected By* property. □

The proof presented above assumes two existing systems. However, the same proof also applies for one existing system, i.e. $\Sigma^a = \Sigma^b$. The effect of the connect tactic on either two different systems or one system is the same.

5.2.2 Disconnect Tactic

The disconnect tactic preserves the *Protected By* property.

Theorem 2 *Disconnect tactic will always preserve the Protected By property*

PROOF. Let $\Sigma = \langle C, N, S, G \rangle$, and Σ has the *Protected By* property by the inductive hypothesis.
Assume there is a connection $n \in N$, where $n = \langle c_i, c_j \rangle$ and $c_i, c_j \in C$.
Disconnecting c_i from c_j means

$$\Sigma' = \langle C, N \setminus \{\langle c_i, c_j \rangle\}, S, G \rangle \tag{5.2}$$

In order to prove Theorem 2, I need to identify all the possible cases where disconnect tactic can be applied and to prove that disconnect preserves the property in each of these cases. A connection has two parameters, i.e. source component (c_i) and target component (c_j). For the components, c_i and c_j, there are three possible cases to consider for disconnect tactic:

- c_j is not a secret. *i.e.* $c_j \notin S$

- c_j is a secret and c_i is allowed to have access to c_j. i.e. $(c_j \in S) \wedge (c_i \in G_{c_j})$
- c_j is a secret and c_i is not allowed to have access to c_j i.e. $(c_j \in S) \wedge (c_i \notin G_{c_j})$

For the first two cases, I remove the connection $\langle c_i, c_j \rangle$ as shown in Function definition (5.2). Removing a connection does not break the *Protected By* property as each secret will still be *Protected By* its granted set. For the third case, a connection where the target component is a secret and the source component is not allowed to have access to that secret should not exist in an acceptable system. If such a connection exists, the system does not have the *Protected By* property.

Removing a connection does not break the *Protected By* property. As I start with a system Σ that has the *Protected By* property, the resulting system, Σ', after applying the disconnect operation maintains its *Protected By* property. □

5.2.3 Create Tactic

The create tactic preserves the *Protected By* property.

Theorem 3 *Create tactic will always preserve the Protected By property*

PROOF. Let $\Sigma = \langle C, N, S, G \rangle$, and Σ has the *Protected By* property by the inductive hypothesis.

Creating a new component c_i means

$$\Sigma' = \begin{cases} \langle C \cup \{c_i\}, N, S, G \rangle & c_i \text{ is not a secret} \\ \langle C \cup \{c_i\}, N, S \cup \{c_i\}, G \oplus \{c_i \mapsto \{\textit{components granted to } c_i\}\} \rangle & c_i \text{ is a new secret} \end{cases} \tag{5.3}$$

In order to prove Theorem 3, I need to identify all the possible cases where create tactic can be applied and to prove that create preserves the property in each of these cases. There are two cases for creating a component: 1) creating a new non-secret component; 2) creating a secret component; The difference between creating the two

types is that creating a secret component will also create a new set which contains all the components that are allowed to have access to that secret.

Function definition (5.3) provides the outcome of all the possible two cases to consider for the create tactic. Creating a new component of any of the two types does not break the *Protected By* property because the new component does not have any connections into and out of it. As I start with a system (Σ) that has the *Protected By* property, the resulting system (Σ') after applying the create tactic will maintain its *Protected By* property. □

5.2.4 Delete Tactic

The delete tactic preserves the *Protected By* property.

Theorem 4 *Delete tactic will always preserve the Protected By property*

PROOF. Let $\Sigma = \langle C, N, S, G \rangle$ and Σ has the *Protected By* property by the inductive hypothesis.

Assume there is a component c_i, where $c_i \in C$.

Deleting a component c_i means

$$\Sigma' = \begin{cases} \langle C \setminus \{c_i\}, N, S \setminus \{c_i\}, G \setminus \{c_i \mapsto \textit{components granted to } c_i\} \rangle & c_i \in S \\ \langle C \setminus \{c_i\}, N, S, G \rangle & c_i \notin S \end{cases} \tag{5.4}$$

$$\text{subject to } \langle c_i, c \rangle \notin N \wedge \langle c, c_i \rangle \notin N \mid \forall\, c \in C$$

In order to prove Theorem 4, I need to identify all the possible cases where delete tactic can be applied and to prove that delete preserves the property in each of these cases. The delete tactic requires one parameter: the component to be deleted, c_i.

There are two cases for deleting a component as shown in Function definition (5.4): 1) deleting a new non-secret component; 2) deleting a secret component. The difference

between the two cases is that deleting a secret component will also delete the set of components allowed to have access to the secret. Deleting a component of any of the two types does not break the *Protected By* property as the delete tactic requires that the component be isolated before deletion is allowed. As I start with a system (Σ) that has the *Protected By* property, the resulting system (Σ') after applying the delete tactic will maintain its *Protected By* property. □

5.2.5 Grant Tactic

The grant tactic preserves the *Protected By* property.

Theorem 5 *Grant tactic will always preserve the Protected By property*

PROOF. Let $\Sigma = \langle C, N, T, S, G \rangle$ and Σ has the *Protected By* property by the inductive hypothesis.

Assume there are two components, c_i and c_j, where $c_i, c_j \in C$.

Granting a component c_i for c_j means creating

$$\Sigma' = \langle C, N, T, S, G \oplus \{c_j \mapsto G_{c_j} \cup \{c_i\}\} \rangle, \text{ where } c_j \in S \tag{5.5}$$

In order to prove Theorem 5, I need to identify all the possible cases where grant tactic can be applied and to prove that grant preserves the property in each of these cases. A connection has two parameters, i.e. source component (c_i) and target component (c_j). For the components, c_i and c_j, there are three possible cases where grant tactic can be considered:

- c_j is a secret and c_i is allowed to have access to c_j
- c_j is a secret and c_i is not allowed to have access to c_j
- c_j is not a secret

For the first two cases, the effect of the grant tactic is as shown in Function definition (5.5). In both these cases, the grant tactic preserves the *Protected By* property because there is no new connection to c_j created and given that the system has the *Protected By* property to begin with, granting a component to have access to a secret does not break the *Protected By* property. The last case, in which c_j is a non-secret component, is not a valid condition to apply the grant tactic. □

5.2.6 Revoke Tactic

The revoke tactic preserves the *Protected By* property.

Theorem 6 *Revoke tactic will always preserve the Protected By property*

PROOF. Let $\Sigma = \langle C, N, T, S, G \rangle$ and Σ has the *Protected By* property by the inductive hypothesis.

Assume there are two components, c_i and c_j, where $c_i, c_j \in C$.

Revoking a component c_i from c_j means

$$\Sigma' = \langle C, N, T, S, G \oplus \{c_j \mapsto G_{c_j} \setminus \{c_i\}\} \rangle, \text{ where } c_j \in S \tag{5.6}$$

In order to prove Theorem 6, I need to identify all the possible cases where revoke tactic can be applied and to prove that revoke preserves the property in each of these cases. Revoke tactic has two parameters, i.e. source component (c_i) and secret component (c_j). For the components, c_i and c_j, there are four possible cases to consider for the revoke tactic:

- c_j is a secret and c_i is not allowed to have access to c_j
- c_j is a secret, c_i is allowed access to c_j, and there is no connection from c_i to c_j
- c_j is a secret, c_i is allowed access to c_j, and there is a connection from c_i to c_j
- c_j is not a secret

The effect of the revoke tactic for the first two cases is as shown in Function definition (5.6). These two cases preserves the *Protected By* property as the effect of the tactic does not introduce a new connection from a component not allowed access to a secret, c_j, to the secret. Given that the initial system, Σ, has the *Protected By* property, the resulting system will also has the *Protected By* property. Thus, the revoke tactic preserves the *Protected By* property. In the third case, revoke tactic is not allowed as it is restricted by the restriction defined in 10. The last case, in which c_j is a non-secret component, is not a valid case to consider using the revoke tactic. □

5.2.7 Sequences of Tactics

An empty system (i.e. system with no components) has the *Protected By* property. This is because, in an empty system, there is no connection to a secret from a component not in the granted set of that secret. I have shown that each of the primitives preserves the *Protected By* property, so by induction, any sequence of applications of the primitives will result in a system that has the *Protected By* property.

5.3 Higher-level Composition Tactics

Here, I introduce examples of higher-level composition tactics, which are derived from multiple applications of the primitives that are introduced in Section 5.1. These derived tactics preserve the *Protected By* property as the primitives have been proven to preserve the *Protected By* property in all cases. The two tactics are proxy and replace. Proxy inserts a new component in between two connected components. This can be achieved through multiple application of the connect and disconnect tactics (i.e. connect the new component to the two existing components and delete the existing connection between the two components). Replace substitutes a component with another component,

which is referred to as the replacing component. This can be achieved through multiple application of connect, disconnect and delete (i.e. connect the existing source component(s) to the replacing component, disconnect and then delete the target component from the source component).

Algorithm 5.1: Proxy Tactic

Data: $\Sigma = \langle C, N, T, S, G \rangle$

Input: *source*, *target* and *proxy*

1 Let source be *src*;

2 **begin**

3 disconnect source from target, i.e. $\Sigma' = \langle C, N \setminus \langle src, target \rangle, T, S, G \rangle$;

4 connect source to proxy, i.e. $\Sigma' = \langle C, N \cup \langle src, proxy \rangle, T, S, G \rangle$;

5 connect proxy to target, i.e. $\Sigma' = \langle C, N \cup \langle proxy, target \rangle, T, S, G \rangle$;

6 **end**

7 Result = $\Sigma' = \langle C, N \setminus \langle src, target \rangle \cup \langle src, proxy \rangle \cup \langle proxy, target \rangle, T, S, G \rangle$

8 Note: the tactic restrictions need to be checked at each usage

The Proxy tactic requires three parameters: *source* component, *target* component and *proxy* components. First, the proxy tactic disconnects the connection, n, between *source* and *target*, where $n = \langle source, target \rangle$ and $source,\ target \in C$. Therefore, the intermediate system can be defined as $\Sigma' = \langle C, N \setminus \langle source, target \rangle, T, S, G \rangle$. Then, the proxy tactic checks the type of both *source* and *proxy* to determine whether the connect primitive is permitted, given the restriction specified in Section 5.1.1. If the connect primitive is permitted, a new connection, n, is created between source and proxy, where $n = \langle source, proxy \rangle$ and $proxy \in C$. The intermediate system can be defined as $\Sigma' = \langle C, N \setminus \langle source, target \rangle \cup \langle source, proxy \rangle, T, S, G \rangle$. Finally, the proxy tactic creates a new connection from *proxy* and *target*, if the connect primitive is permitted. The final

system is then $\Sigma' = \langle C, N \setminus \langle source, target \rangle \cup \langle source, proxy \rangle \cup \langle proxy, target \rangle, T, S, G \rangle$. The algorithm for the proxy tactic is shown in Algorithm 5.1.

The condition for proxy is:

> For all secret nodes, d, where $d \in S$ and $(c_b = d)$ or $(c_b \in (G^a \cup G^b)_d)$, then:
> $c_a \in (G^a \cup G^b)_d \wedge ((c_a \in (T^a \cup T^b)) \vee (\forall c. \langle c, c_a \rangle \in N \rightarrow c \in (T^a \cup T^b) \wedge c \in (G^a \cup G^b)_{c_b}))$

The Replace tactic requires two parameters: the *replacing* component and the *target* component. First, I determine all the components that hold an access to the *target* component. For each of these components, it is connected to the *replacing* component and is disconnected from the *target* component. Then, the *target* component is deleted if it is isolated after the connect and disconnect primitives have been applied. The algorithm for the replace tactic is shown in Algorithm 5.2. The conditions for replace are inherited from the connect tactic.

5.4 Composing Design Fragments to Support an Assurance Case

When building an assurance case, I determine the security property that the system needs to satisfy and set that as the root claim. Then, I identify potential attacks or loopholes and make claims about how the system remains secure despite these possible attacks. These claims are the subclaims of the root claim. These claims are then supported with arguments organised according to the structure of the architecture and through evidence provided by analyses.

When there is a security violation in the proposed system, verified capability-specific design fragments are composed with the system to mitigate that attack. After this, the

security properties of the composite design are analyzed. The verification of the composite design feeds into the construction of the assurance case as evidence that supports the claims about system security.

Algorithm 5.2: Replace Tactic

Data: $\Sigma = \langle C, N, T, S, G \rangle$

Let the *replacing* component be *rep* and *target* component be *tar*;
foreach *connection, n, in the connection set N* **do**
    $n = \langle a, b \rangle$;
    if $tar \in n$ **then**
        if $tar == a$ **then**
            disconnect *tar* from *b*, i.e. $\Sigma' = \langle C, N \setminus \langle tar, b \rangle, T, S, G \rangle$;
            connect *rep* to *b*, i.e. $\Sigma' = \langle C, N \cup \langle rep, b \rangle, T, S, G \rangle$;
        end
        else tar == b
            disconnect *a* from *tar*, i.e. $\Sigma' = \langle C, N \setminus \langle a, tar \rangle, T, S, G \rangle$;
            connect *a* to *rep*, i.e. $\Sigma' = \langle C, N \cup \langle a, rep \rangle, T, S, G \rangle$;
        end
    end
end
if *isIsolated(tar)* **then**
    delete *tar*, i.e. $\Sigma' = \langle C \setminus \{tar\}, N, T, S, G \rangle$
end
Note: the tactic restrictions need to be checked at each usage

Chapter 6

Verification Procedures

In this chapter, I consider how to analyse the security properties of capability-based design fragments and application designs.

Each design fragment has associated verification procedures. A verification procedure is a set of statements that is used to check security properties of a system. In general, various analysis techniques can be used for verification procedures.

Verifying individual design fragments gives some assurance about the security properties of the fragments. Furthermore, it helps identify localized problems in an individual design fragment before those problems propagate to the whole application design through composition. Since these fragments are composed together to form the application design, analysis needs to be performed on the whole application as well as on individual fragments. The composition tactics, defined in Section 5.1, are proven to preserve security proterty assuming that the system has the security property to begin with. If a system does not yet have the *Protected By* property, performing the analysis is required. This is crucial because analysing the composite design indicates that it retains the intended security properties, despite transformations introduced in design or composition. The analysis performed on these designs will not only provide feedback to

improve the designs but also provides evidence about the security properties.

In this work, I use Points-To analysis using Binary Decision Diagrams (BDD) (Berndl et al., 2003) to verify the security properties of both the design fragments and the application design resulting from the composition of design fragments. Section 6.1 will detail how Points-To analysis is used in the context of a SAM model. I then define and explain what a verification procedure is in Section 6.2. I show my proposed template to express security properties and its mapping to elements of a verification procedure in Section 6.3 before concluding with a discussion in Section 6.4.

6.1 Points-To analysis using Binary Decision Diagrams (BDD) as Database Queries

Points-To analysis using Binary Decision Diagrams (BDD) establishes which pointers can point to which variables and has been shown to scale for analysis of large programs (Berndl et al., 2003). Points-To analysis over-approximates the actual behavior of a program, which means that a pointer might (but may not ever) point to a particular variable. This means that a pointer might point to a particular variable in the analysis even if in reality this may not happen in the system. It is conservative and considers the worst-case scenario. Berndl et al. (2003) has implemented Andersen's subset-based Points-To analysis (Andersen, 1994), which is flow and context-insensitive. Flow insensitive means that the analysis does not take into consideration the order of program statement execution. Context insensitive analysis ignores the context in which the execution of a program statement occurs. In order to allow for natural expressiveness and support context-sensitiveness, Whaley (2007) used a declarative language, Datalog (Ceri et al., 1989), and developed a Datalog engine (`bddbddb`) to analyze computer programs with Points-To analysis and BDD. The analysis that is implemented in SAM is inspired by

`bddbddb`. Thus, it is a flow-insensitive analysis with support for context sensitivity.

6.2 Verification Procedure

A verification procedure is a set of statements that is used to check security properties of a system. It has five main elements: name, procedure (statements), a set of source components, a set of target components, and a set of exclusions (optional). A name is an identifier of a verification procedure. The components in the exclusion set are trusted components that are allowed to have access to the target components and thus are exempted from the analysis. I check whether a source component has an access to a target component (secret), excluding the trusted components that are allowed to have access to a secret, utilizing the three sets of components. Procedures are statements that express the security property to be checked. The representation of the statements depends on the choice of analysis technique.

In SAM, Points-To analysis establishes which components point to (i.e. have access to) other components. This generates a mapping of which components have access to other components. This includes propagating accesses during execution, based on the behavior of each components that are involved. Security goals are specified as Datalog rules. A Datalog rule consists of a rule head and a rule body. A rule head consists of a name and a set of parameters for the rule. A rule body contains statements that define the rule. First, the program behaviors in a SAM model (specified in Java-like language) are translated to Datalog. This translation is provided and performed by SAM. Then, facts are deduced from Datalog rules (i.e. the security goals) that are specified in the SAM model. Finally, I check the Datalog query, using the mapping generated by the Points-To analysis to check whether there exists a capability (direct access) from a component to another component. Checking the Datalog query ensures that the design fragment

satisfies its desired security properties.

Let us consider a security goal where a system is secure if a component *meter* (source) does not have access to *logFile* (target). The `hasRef(source,target)` predicate that is provided by SAM can be used to check whether *meter* has access to *logFile*. As there are no exclusions, this predicate is sufficient to express the intended security goal. I then check whether the security goal is checking that the source does not have access to the target, `assert !hasRef(`*`meter`*`,`*`logFile`*`)`. This will only succeed if it is not inconsistent with the existing fact base.

More complicated security goals with multiple source components, target components and exclusions can also be specified. This will be shown in the next section.

6.3 Security Property Template

I define a template for describing a security goal of a system to make it easier to specify the verification procedures to check the intended security goal. The template requires three parameters, namely *source*, *secret*, and *exclusions*. *Source* represents a set of components to check on. *secret* is a set of components that needs to be protected while *exclusions* is a set of components that are trusted to have access to the secret. The template is as follows:

Source no access to *secret*, except *exclusions*

This template is then translated to a Datalog rule. The translated Datalog rule looks at direct access to secrets, only after all possible accesses in the system have been propagated. I rely on the Points-To analysis in SAM to propagate the accesses and assume that the propagation is correct, i.e. maximum access propagation that is possible based on the system's behavior.

As discussed in the previous section (Section 6.2), a verification procedure has five

elements: name, procedures, a set of source components, a set of target components, and a set of exclusions (optional). In order to translate the template into a verification procedure, a mapping between the parameters in the template and the elements of a verification procedure is created. *Source*, *secret* and *exclusions* of the template map directly to those in the verification procedure. The name of the verification procedure can be auto-generated. The number of parameters that the Datalog rule requires are then determined. This can be calculated by the number of *secret* ($|\{secret\}|$) + 1 (for source).

Next, the content of the Datalog rule body is determined by using the information about exclusions and targets. For every exclusion, the negation of the `MATCH(a, b)` predicate that is implemented in SAM is used. This predicate tests whether *a* is equal to *b*. For every target, the `hasRef(a,b)` predicate is used to check whether *a* has access to *b*.

Finally, the assertions are written to ensure that all the components in the system, excluding the exclusions, have no access to the targets. The negation of the Datalog rule is iteratively checked to be true for each source.

Figure 6.1 shows a generic verification procedure that is generated from my security property template. Lines 2-4 exclude the trusted components from the check and Lines 5-7 check whether the source has access to the secrets. Note that '`,`' here represents logical conjunction. Lines 8-10 assert that the security property is not breached for each source. A parameter that starts with '`?`' is a Datalog variable.

Consider the security goal (SG-1) where a system is secure if *meter* does not have access to *logFile*. This goal can be captured in the template as *`meter`* `no access to` *`logFile`*`, except none`. This can be translated into a verification procedure and can be named as `accessMeterlogFile`. Then, the number of parameters that the Datalog rule requires are determined. As there is only one element in the target component set, the Datalog rule requires two parameters (one parameter for source). There

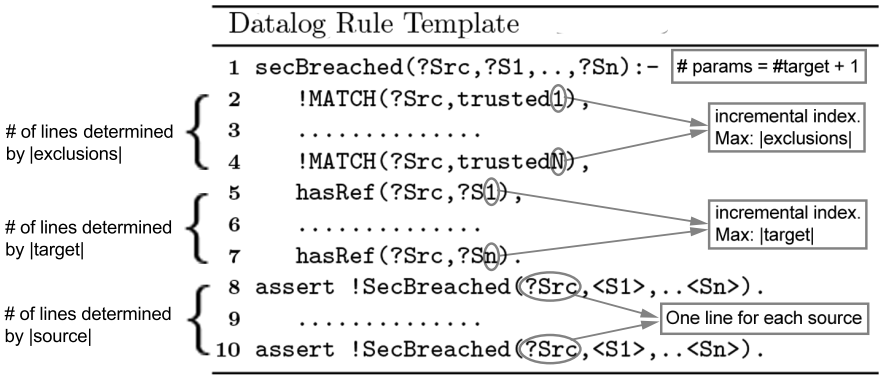


Figure 6.1: Verification Procedure from template

are no exclusions in this verification procedure. The resulting verification procedure is shown in Listing 6.1.

Listing 6.1: SG-1 in Datalog

```
1 declare accessMeterlogFile(Ref Src, Ref S1).
2 accessMeterlogFile(?Src,?S1):-
3    hasRef(?Src,?S1).
4 assert !accessMeterlogFile(<meter>,<logFile>).
```

A more complicated security goal (SG-2) will be that all components in the system do not have access to *logFile*, except *logger* and *logManager*. First, "accesslogFile" is assigned as the name of the verification procedure. There is only one target component and thus the Datalog rule requires two parameters. A Datalog variable (starts with '?') is used for the source component parameter. This variable represents a component in the system. The Datalog engine will iteratively deduce facts from the specified Datalog rule, `noAccesslogFile`, by querying each component in turn. The verification procedure is shown in Listing 6.2.

I also define a generic Datalog representation of the *Protected By* property for SAM, as shown in Listing 6.3. In this Datalog rule, I assume that the access propagation is done by the Points-To analysis in SAM and thus is not represented in the rule. Lines 1-4 de-

Listing 6.2: SG-2 in Datalog

```
1 declare accesslogFile(Ref Src, Ref S1).
2 accesslogFile(?Src,?S1):-
3     !MATCH(?Src,<logger>),
4     !MATCH(?Src,<logManager>),
5     hasRef(?Src,?S1).
6 assert !accesslogFile(?Src,<logFile>).
```

clare new predicates that are required to express the *Protected By* property. `isSecret` tags a component as a secret and `isGranted` maps a component, which is allowed to have access to a secret, to a secret. `isValidGranted` filters facts captured by `isGranted`, only retaining facts where `S` is a secret. The `isProtectedBy` rule checks that for each secret, there is no other component, apart from those which are granted access to a secret, which has access to a secret.

Consider a system that calculates checksum, which is shown in Figure 6.2a. The *client* invokes the *calculateChecksum* function of the *orchestrator* and sends the data to be calculated on. Upon receipt, the orchestrator invokes the checksum function of the *checksummer*, who then stores the result into *checksumStore*. The secret that needs to be protected is the *checksumStore* and only *checksummer* is allowed to have access to it. Both *client* and *orchestrator* are trusted.

Using the Datalog rule for the *Protected By* property in Listing 6.3, only two additional lines are required:

- `isSecret(<checksumStore>)` — specifies that *checksumStore* is a secret
- `isGranted(<checksummer>,<checksumStore>)` — specifies that *checksummer* is in the granted set of *checksumStore*.

I can then write `assert !isProtectedByBreached(?Src,?S)` to ensure that the *Protected By* property is not breached.

Listing 6.3: Generic Datalog Rule for the *Protected By* property (access propagated)

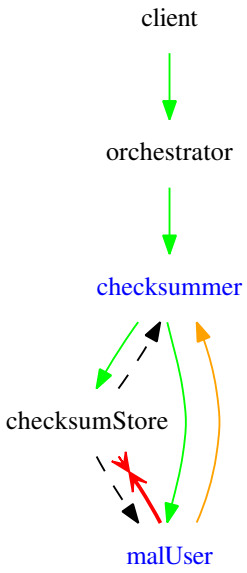
```
1 declare isSecret(Ref object).
2 declare isGranted(Ref object, Ref secret).
3 declare isValidGranted(Ref Src,Ref S).
4 declare isProtectedByBreached(Ref Src, Ref S).
5 isValidGranted(?X,?S):-
6    isSecret(?S),
7    isGranted(?X,?S).
8 isProtectedByBreached(?Src,?S):-
9    !MATCH(?Src,?S),
10   !isValidGranted(?Src,?S),
11   isSecret(?S),
12   hasRef(?Src,?S).
```

A malicious (untrusted) user that has an access to the *checksummer* is then introduced, as shown in Figure 6.2b. The *Protected By* property is breached as *malUser* has access to *checksumStore* and is not in the granted set of *checksumStore*. The red arrow from *malUser* to *checksumStore* signifies a security violation while the orange arrow from *malUser* to *checksummer* shows the cause of the violation.

6.4 Discussion

Reusing design fragment verification procedures for the application design might require their modification. This modification is the effect of applying the composition tactic as defined in Chapter 5. This is done to reflect the goals of the application, which might differ from those of the individual design fragments. Such modification is done during

(a) Baseline – All the arrows, except for the black dotted arrow from checksumStore to checksummer, are in green which means that the functions of each component has been invoked. A black arrow signifies that checksumStore does not invoke any functions of checksummer. checksummer is an untrusted component (in blue color) that is allowed to have access to checksumStore.



(b) with malicious (untrusted) user, shown in blue. The red arrow (from malUser to checksumStore) shows a security violation while the orange arrow (from malUser to checksummer) shows the cause of the violation. The dotted arrows out of checksumStore are in black, which signifies that checksumStore does not invoke any functions of malUser and checksummer. The other arrows are in green, which means that the functions of each component has been invoked.

Figure 6.2: Checksum Calculator

composition. For instance, if the goal of a design fragment is to prevent unauthorized access to the encryption key while the goal of the application is to prevent unauthorized access to the encryption key and encrypted file at the same time, the goal of the application subsumes that of the design fragment. The Datalog rule needs to be modified to check for access to both key and file simultaneously. The goal of the application can then be written, using the template defined in Section 6.3, as follows:

`All no access to key,file, except encryptedStorage`

This is then translated to into Datalog rule for analysis to ensure that the goals of

the application are satisfied. Listing 6.4 shows the verification procedure for checking the application. The underlined texts are the differences between this verification procedure and the encrypted storage verification procedure. Line 4 excludes *file* from being checked and line 7 checks whether a component has access to *file*.

Listing 6.4: Application Security Goal in Datalog

```
1 declare isSecBreached(Ref Src,Ref T,Ref T1).
2 isSecBreached(?Src,?T,?T1):-
3     !MATCH(?Src,?T),
4     !MATCH(?Src,?T1),
5     !MATCH(?Src,<encryptedStorage>),
6     hasRef(?Src,?T),
7     hasRef(?Src,?T1).
8 assert !isSecBreached(?Src,<key>,<file>).
```

One possible limitation to the template, which is defined in Section 6.3, is that I rely on users to follow the principle of least privilege when allowing access to a secret, i.e. only allow a component to have access to a secret if it is essential for the system. Components that are granted access to a secret can be thought to be "internal" to the system design — not directly exposed to attack.

Chapter 7

Evaluation

In this chapter, I evaluate my approach using two case studies of different domains, which are presented in two separate sections. Section 7.1 evaluates the applicability of the composition approach and the expressiveness of the six composition primitives to harden an existing system and to extend its functionalities. Verified capability-based design fragments are applied to a generic Continuous Deployment (CD) pipeline using the composition primitives to harden the security of the pipeline. The pipeline is then verified to be secure in the presence of various attacks. The approach is shown to be feasible to be applied in the context of a CD pipeline and to secure the pipeline. Section 7.2 evaluates the effectiveness of the higher-level composition tactics to design a secure smart meter, based on industrial requirements. Starting with a simple model, the composition tactics are applied to compose capability-based design fragments with the model in order to be withstand different attacks. The approach is shown to be feasible to be applied in a different domain and that the composition tactics are sufficient to express the necessary compositions.

7.1 Continuous Deployment Pipeline

In this section, I evaluate the feasibility of applying my pattern-based composition approach and the expressiveness of the composition tactics using a Continuous Deployment (CD) pipeline. I define the possible threats that I consider in a threat model and evaluate whether my approach can sure a CD pipeline in the presence of these threats. I also assess whether the composition primitives are sufficient to express different compositions that are required to secure the pipeline.

Many companies have embraced the concept of Continuous Deployment (Bass et al., 2014), which aims to deploy code changes to a production environment multiple times a day. Each change automatically goes through a set of tools that perform activities like integration build, deployment (and testing) to various testing environments, and deployment to production environments. The tool chain performing these activities is referred to as a Continuous Deployment (CD) pipeline.

A key challenge in a CD pipeline is the security of the pipeline itself (Bass et al., 2014, chap. 8.1). First, different roles in the development team and the operation team should have different access to different parts of the pipeline. For example, a developer should not be able to deploy to production directly without her changes going through the pipeline. Certain build and test jobs can only be triggered by certain roles. Second, the testing and production environment should have total isolation. Major real world outages have happened because a component in the testing environment is accidentally connected to production database (Bass et al., 2014, chap. 2.3.). Third, a compromised or misconfigured continuous deployment pipeline may have malicious code or unwanted debugging/experimental code that ends up being deployed to production. A typical CD pipeline is not designed with all the above security requirements in mind. Thus, the aim of the case study is to use my approach to enhance the security design of a CD pipeline satisfying the security properties derived from the above requirements.

In order to make this work practical, I work under two real-life constraints. Firstly, formally verifying, from scratch, all the components and systems that I use is infeasible. Secondly, I have to work with existing components to ensure that the changes are minimal. In this work, I trust some specialised components, which can then be formally verified. Securing a real-life system using formal or semi-formal techniques is both challenging and necessary to make these techniques usable. The target platform is the Amazon Web Services (AWS) and security model is capability-based. AWS provides assurances about the correctness of their systems through formal verification and model checking (Newcombe et al., 2015). Although the AWS implementation has not been formally verified, I treat its security mechanisms as trusted because of real-life constraints.

Section 7.1.1 describes background information of a Continuous Deployment pipeline. I outline the existing security mechanisms that can be utilised in a CD pipeline in Section 7.1.2. Then, I formulate a threat model to be considered for the pipeline in Section 7.1.3. I then describe the process of securing a continuous deployment pipeline by applying my approach in Section 7.1.4 before concluding with a discussion in Section 7.1.5.

7.1.1 Background

Continuous Deployment pipelines vary from one company to another, depending on their existing practices. However, each of these pipelines has a common sequence of stages, which include building the code, testing the code and deploying the code to production. Each stage may require different tools and shares some commonly used tools. Figure 7.1 shows the common stages that are performed in a Continuous Deployment pipeline. It starts when a developer commits code into a code repository. This commit will trigger a build server, which monitors the code repository, to perform an integration of the newly committed code and the existing code and build the code into application binary. The

server will then perform integration tests on the application binary.

Pulling the code, building the application binary from the code and performing integration tests are part of Continuous Integration (CI). Continuous Integration can be described as progressing through the pipeline by means of automation until the integration tests are performed. This differs from Continuous Deployment where the automation helps progress to deployment of an application to production environment.

In this case study, Jenkins[1] is chosen because it is the current state-of-art CI server. It is an open-source application for continuously building and testing software applications. It is typically used as a build server that performs and orchestrates several steps in a CD pipeline, which include pulling the source code, building application binary from source code, running the test suites, packaging the application binary into an image and storing the image into a repository or storage. Storage uses AWS Simple Storage Service (S3) Buckets[2].

In the application building stage, the code is built and its binary is then packaged, also called the build artifact, into an image. Packaging binaries into an image helps to preserve consistency throughout the pipeline (i.e. from testing environment to production environment). The consistency that is of concern is the environment that the application is running on, which includes the software dependencies and configurations

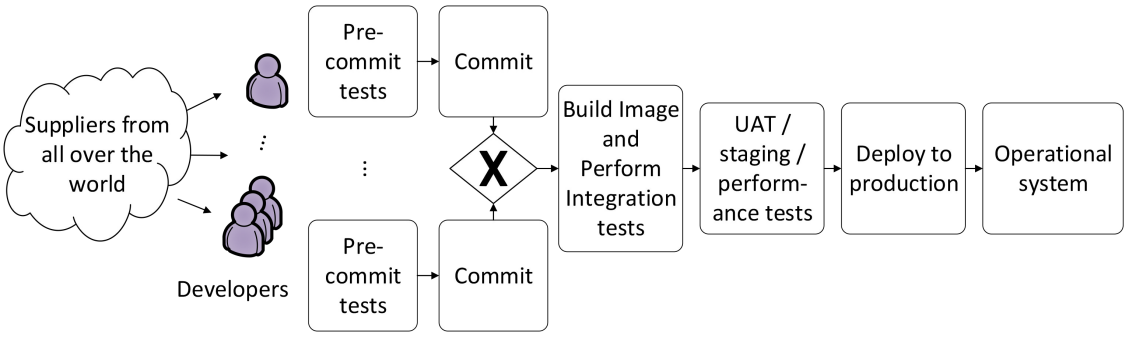


Figure 7.1: Generic Continuous Deployment pipeline (Bass et al., 2014)

[1]Jenkins—http://jenkins-ci.org/

[2]S3—http://aws.amazon.com/s3/

that are required by the application. This image is stored in a storage, such as AWS S3.

There are different types of tests required during the testing stages. These include unit tests, integration tests and end-to-end tests of various lengths. They are often run inside different environments. The testing environment is set up and the tests are then run on the application image to ensure that the application demonstrates its intended functionalities. A deployer is required to setup the various testing environments, which includes installing the application and its dependencies inside a virtual machine, configuring and running the application, and triggering the tests. AWS Opsworks[3] is a service that helps to set up the environment and to install the application. It handles the provisioning of resources (through AWS APIs) when setting up the environment and uses Opscode Chef[4] to configure environment inside a Virtual Machine (VM).

If the application image passes all the tests, it will then be deployed to the production environment. Usually, a release manager, who is a human operator, has to approve the deployment of a particular image into the production environment. Upon approval, a deployer will then deploy the image to the production environment.

There are two popular strategies for deploying applications (Bass et al., 2014), namely big flip and rolling upgrade. In a big flip strategy, N VM instances are provisioned to run a new version of the application, V_B, while N VM instances are running the current version of the application, V_A. Once all the instances running V_B are provisioned, the instances running version V_A are terminated.

In a rolling upgrade, a small number of k instances at a time currently running version V_A are taken out of service and replaced with k instances running version V_B. This is repeated until all the instances have been upgraded and are running version V_B. A virtue of rolling upgrade is that it only requires a small number of additional instances to perform the upgrade. Rolling upgrade is a widely used method for upgrading instances

[3]Opsworks—http://aws.amazon.com/opsworks/

[4]Chef — https://www.chef.io/chef/

(Dumitraş and Narasimhan, 2009).

7.1.2 Existing Security Mechanisms

There are several security mechanisms that can be utilised in the pipeline, namely from the platform (AWS in this case), operating system and the build server - Jenkins. AWS Identity and Access Management (IAM)[5] security mechanism binds authority to users and roles. In IAM, each user and each role is associated with at least one security policy. This security policy defines what actions can be performed by the holder of the policy on specific AWS resources. Initially, AWS (implicitly) denies all actions on all resources. Specifying in a policy that its holder has the right to perform an action on a particular resource, e.g. write object to a specific S3 Bucket, takes precedence over the implicit deny. An explicit deny takes the highest precedence, i.e. explicitly denying an action will override any policies that allow the same action on the same resource.

Furthermore, IAM also provides the ability to change or assume roles at runtime. Assuming a role creates a security credential tuple, consisting of an AWS access key id, AWS secret access key and temporary security token, which is similar to a capability since it can be passed to other instances granting them new access rights.

An EC2 instance[6] running as a particular user or assuming a particular role will have the corresponding authority to access resources or invoke AWS operations. An EC2 instance is a virtual machine running on top of AWS infrastructure.

There are three characteristics of a capability: 1) it provides access to an object in a system; 2) it can be passed around; and 3) it is held by a principal or user. The first two characteristics are provided by the security credential tuple in AWS, which is created when assuming a role. AWS security model assigns the rights to a user or a role and thus

[5]IAM—http://aws.amazon.com/iam/

[6]EC2—http://aws.amazon.com/ec2/

aggregating access at the principals or users. Note that the AWS security model is not explicitly a capability-based model, however, it has the characteristics of such a model, and so my approach and design fragments map well to it. In SAM models of Continuous Deployment pipeline, the components are either running on EC2 instances and/or using AWS resources (such as S3 buckets), and the capabilities are AWS role-based security tokens.

The operating systems that are running on the EC2 instances also provide security mechanisms to govern what operations, such as installing an application, can be performed inside the instance. As mentioned in Section 7.1.1, AWS Opsworks is used to setup the environment inside the instances. Opsworks requires root access in order to perform these operations. For this reason, the security mechanisms offered by the operating system is disregarded as they are overridden by Opsworks' root access.

Jenkins provides authentication and authorization mechanisms. Authentication can be achieved by creating a user database for Jenkins. Matrix-based security is commonly used as Jenkins' authorization strategy. It allows administration of specific pre-defined access rights to users or groups. The full list of pre-defined rights can be found online[7].

Table 7.1 summarizes the AWS services that used in the CD pipeline.

Table 7.1: AWS Services used in the pipeline

AWS Service	**Description**
Elastic Compute Cloud (EC2)	a Virtual Machine (VM) in AWS infrastructure
Simple Storage Service (S3)	a file storage service in AWS infrastructure
Identity and Access Management (IAM)	a service to control access rights to AWS resources and define security policy
OpsWorks	a service that helps to set up the environment and to install the application in EC2 instances.

[7]https://wiki.jenkins-ci.org/display/JENKINS/Matrix-based+security

7.1.3 Threat Model

One question to consider when attempting to secure any software system is how powerful the attacker is. Another question is what are the data or parts of the system that must be protected. With these questions in mind, I have defined a threat model for my Continuous Deployment (CD) pipeline.

Defining a threat model starts with defining the assets (resources) to be protected. The assets in the pipeline are Code bucket, credential bucket, config bucket, image bucket, build server. Then, the potential attackers (malicious users) are identified. A malicious user can call any method of the components that they have access to, with any possible parameters. Furthermore, a malicious user can try to pass around any capabilities they possess to the components they can access.

After defining the abilities of an attacker, different attacks that can happen are considered:

- A remote attacker may attempt to exploit a component in the build environment that is directly accessible from outside of the environment. If successful, an attacker can gain the privileges of the process. I do not consider further privilege escalation (to administrative rights), as this would trivially compromise all processes on the machine.

- A remote attacker may attempt to infiltrate the CI server (Jenkins). If successful, an attacker can bypass the access rights enforcement and gain access to all the assets. Example attacks that can happen include:

  - The attacker can fetch the source code from the repository and modify it. This introduces an exploit which can compromise the entire build process.
  - The attacker can tamper with the build image and goes unnoticed.
  - The attacker can deploy the wrong version of image.

  - The attacker can gain access to the credentials.
  - The attacker can read and modify the configurations.

Finally, the existing countermeasures to the potential attacks and assumptions about the threats are elicited, which include:

- Compiler is correct.
- No attacks on network links both on the public Internet (i.e. on the connections between my machine and AWS) and on AWS infrastructure.
- Jenkins is installed on an EC2 instance.
- Matrix-based authentication and Project-based authentication in Jenkins can be bypassed as shown by Bass et al. (2015). Furthermore, there are known vulnerabilities to bypass intended restrictions in Jenkins as reported in CVE-2014-3663[8] and CVE-2014-2058[9].
- AWS IAM policy can restrict access to the bucket only from the build server.
- Sufficiently strong cryptography is used for encryption.

7.1.4 Securing the Continuous Deployment Pipeline

The high-level security requirements that need to be satisfied are that malicious code is not deployed through the pipeline and that there is no direct communication between components in the testing and production environments. The first high-level security requirement can be broken down into four requirements, one for each of the three stages (see Section 7.1.1) of the pipeline and one requirement about credentials. These requirements are that the malicious user cannot have access (i.e. read, write and grant) to code,

[8] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3663
[9] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2058

the correct version of build image is deployed for testing and is untampered with, the tested image to be deployed to production is untampered with and credentials are not leaked. The ability of the malicious user is defined in Section 7.1.3. An untampered image is one that is the product of a successful execution of a process described in the specification, and that no additional changes are made. These requirements can be further broken down to be more specific.

The second high-level security requirement can be broken down into two requirements, namely that there is no component that has access to both production and testing environments, and that the production environment is isolated from the testing environment. This requirement is specified to prevent components in the testing environment from accessing anything in the production environment which may break the production environment. After identifying these, an assurance case is built (Figure 7.2) in order to capture and demonstrate that the pipeline satisfies its intended requirements. We claim that malicious code is not deployed through the pipeline as the root claim in the assurance case. In order for that claim to be true, all the sub-claims (i.e. the broken down requirements) need to be true as well. The assurance case is presented in Figure 7.2.

We start off by modeling the existing pipeline model without any security consideration and then harden the security of the pipeline. The pipeline starts with a developer making a change in the form of a code commit. Jenkins then pulls that code commit and builds the code. Building the code results in a build artifact, which is an application binary. Jenkins then packages the build artifact into an image and stores it in AWS S3. After that, Jenkins triggers a deployer to deploy the built image to a testing environment for testing. The deployer is a simple application that uses AWS OpsWorks to set up the testing environment, deploy and start the image inside the testing environment. The tests are then triggered. The logs from the test are then stored in an S3 bucket.

In this model, Jenkins is the main orchestrator that has access to *codeBucket*, *creds-*

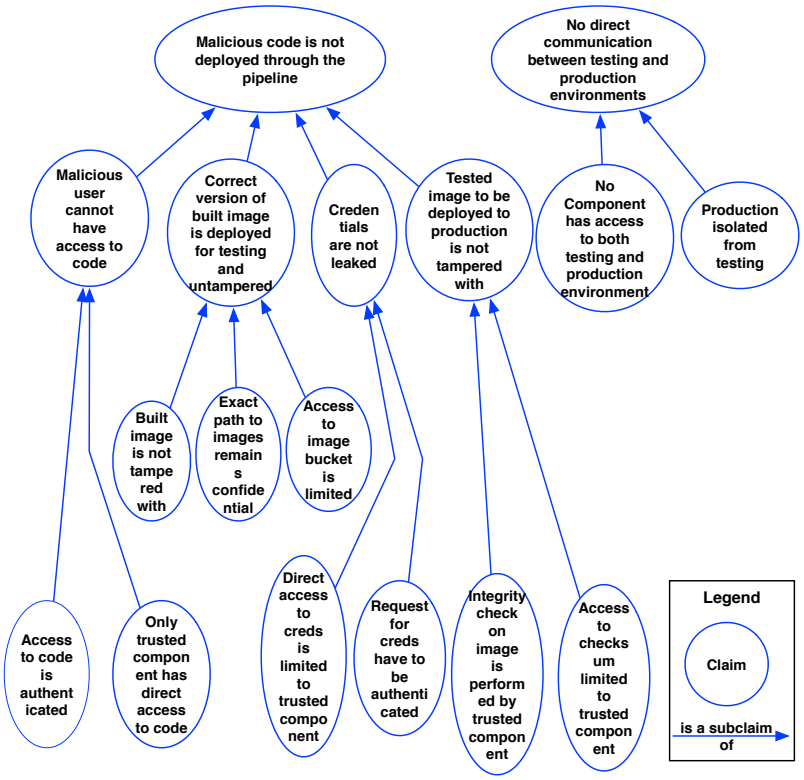


Figure 7.2: Continuous Deployment Assurance Case (subset)

Bucket, *imageBucket*, *configBucket* and *deployer*. *codeBucket*, *credsBucket*, *imageBucket* and *configBucket* are all AWS S3 buckets, which are modelled to have put (write) and get (read) functions. Figure 7.3 shows the initial model of the pipeline. The *codeBucket*, *credsBucket* and *configBucket* are identified as the secrets that need to be protected and only Jenkins (trusted) can have access to them. Furthermore, the *imageBucket* is identified as a secret and trust Jenkins and *deployer* to have access to it. In order to check that these properties are satisfied, two Datalog rules, `bktBreached` and `imgBreached`, are written as shown in Listing 7.1.

The initial model satisfies these rules, with the assumption that Jenkins is a trusted component. Satisfying these rules provides evidence to support a subset of the claims in the assurance case, in particular "only trusted component has direct access to code","only a trusted component has access to an image" and "direct access to credentials is limited

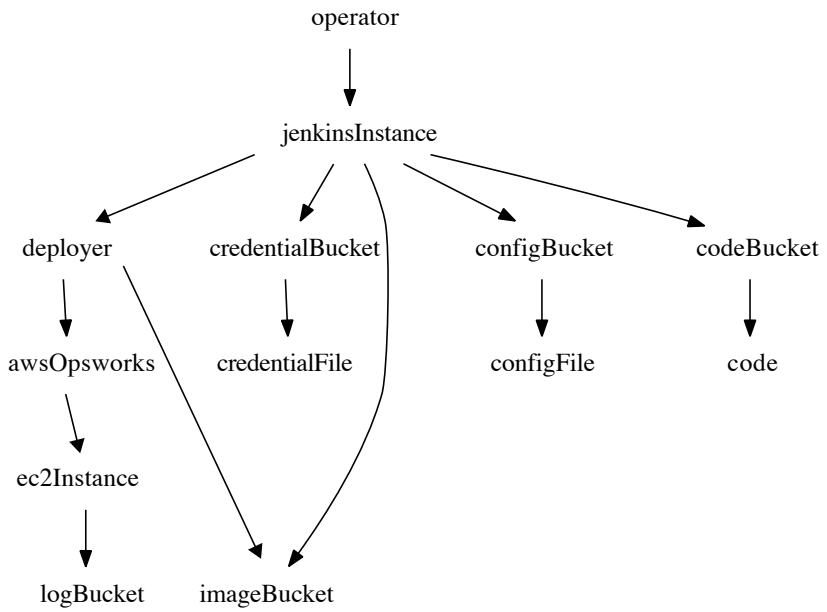


Figure 7.3: Initial design of the CD pipeline. This shows the structure of the initial design of the pipeline, where an arrow pointing to a component represents holding a capability to it.

to a trusted component". One major weakness of this model is that we rely on the assumption that Jenkins is trusted. If Jenkins is infiltrated and thus considered as untrusted, the security properties of the pipeline will not hold, given that Jenkins is the main orchestrator in the model. Therefore, we have to model Jenkins as an untrusted component to ensure that the pipeline is secure even if Jenkins is infiltrated.

Jenkins is modelled as an untrusted component in SAM, whereby an untrusted component may invoke any methods on the components it has access to and tries to pass around any capabilities it possesses to the components it can access. Figure 7.4 shows a model of the continuous deployment pipeline when a malicious user infiltrated Jenkins. The red arrows show the security violations in the model, where malicious user has gained access to *codeBucket*, *credsBucket*, *imageBucket* and *configBucket*.

My process for hardening the security of the pipeline has the following steps:

1. Identify the security requirements for the pipeline.

Listing 7.1: Verification Procedure of the initial design

```
1 bktBreached(?Src,?Target):-
2     !MATCH(?Src,?Target),
3     !MATCH(?Src,<Jenkins>),
4     hasRef(?Src,?Target).
5 imgBreached(?Src,?Target):-
6     !MATCH(?Src,?Target),
7     !MATCH(?Src,<Jenkins>),
8     !MATCH(?Src,<deployer>),
9     hasRef(?Src,?Target).
10 assert !bktBreached(?Src,<codeBucket>).
11 assert !bktBreached(?Src,<credsBucket>).
12 assert !bktBreached(?Src,<configBucket>).
13 assert !imgBreached(?Src,<imageBucket>).
```

2. Identify the trusted and untrusted components of the pipeline.

3. Repeat until all of the requirements have been satisfied OR can no longer decompose the untrusted components:

   (a) Model the interactions between the components.

   (b) Analyze the model to check whether it satisfies its requirements.

   (c) Decompose untrusted components causing an unsatisfied requirement into a trusted and an untrusted portion.

This process is based on the idea that the actual building and deploying activities are small pieces of code that can be encapsulated into trusted components and that the trusted components can mediate access to the actual building and deploying activities.

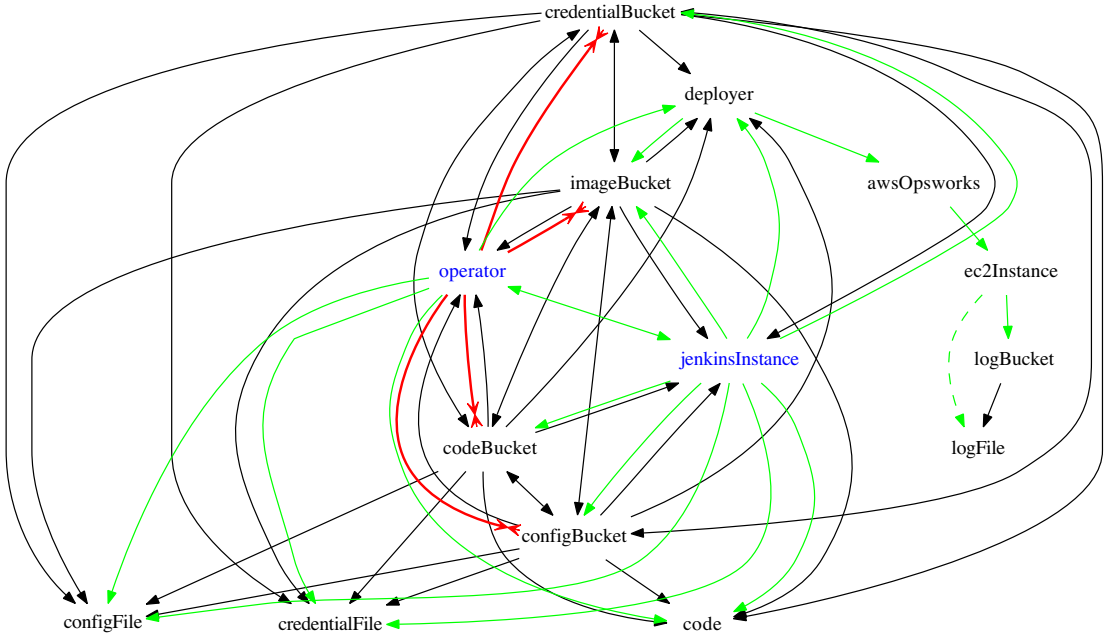


Figure 7.4: Initial design of the CD pipeline with infiltrated Jenkins. This figure shows the model after all the possible accesses are propagated. There are four security violations in this figure, which are represented as four red arrows. These arrows are: *operator* to *imageBucket*, *operator* to *codeBucket*, *operator* to *credentialBucket*, and *operator* to *configBucket*. The *operator* is not allowed to have access to these four components but gained access from the infiltrated *jenkinsInstance*.

More detail about the formal portions of this process can be found in Rimba, Zhu, Bass, Kuz and Reeves (2015).

We aim to harden the pipeline with Jenkins being untrusted. The first step is to add authentication for the component that is retrieving the code, config file and credential file. The current model is composed with the authentication enforcer (Schumacher et al., 2006) design fragments. The authentication enforcer (Figure 7.5) aims to create a single point of access to receive interactions of a subject and verify the identity of the subject. The security property of the authentication enforcer design fragment is that the user store should remain confidential. A Datalog rule, shown in Listing 7.2, is written to check that only *authenticationEnforcer* can have access to *userStore*.

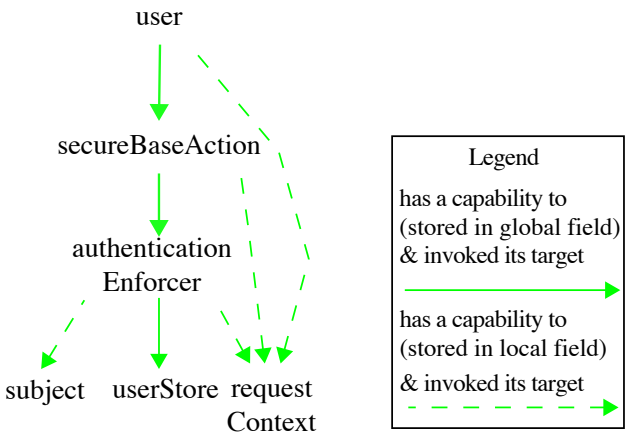


Figure 7.5: Authentication Enforcer Design Fragment. All the arrows are in green, which means that the functions of each component has been invoked.

Listing 7.2: Verification Procedure of the Authentication Enforcer

```
1 uStoreBreached(?Src,?T):-
2     !MATCH(?Src,?T),
3     !MATCH(?Src,<authenticationEnforcer>),
4     hasRef(?Src,?T).
5 assert !uStoreBreached(?Src,<userStore>).
```

We need to insert the authentication enforcer design fragment in between Jenkins, *codeBucket*, *credsBucket* and *configBucket*, in order to moderate their interactions. First, we use the connect tactic, connecting Jenkins to *secureBaseAction*. Then the disconnect tactic is used to detach *codeBucket*, *credsBucket* and *configBucket* from Jenkins. As each of *codeBucket*, *credslBucket* and *configBucket* is a secret, Jenkins is removed from each of their granted component sets using the revoke tactic. Then, *secureBaseAction* is added to the granted set of *codeBucket*, *credsBucket*, and *configBucket* using the grant tactic and connect them. The resulting model is shown in Figure 7.6. The affected Datalog rule (`bktBreached` in this case) is modified every time the tactic is applied. The resulting rule is shown in Listing 7.3. We verify that the composite design satisfies the `uStoreBreached`, `imgBucketBreached` and `bktBreached` rules.

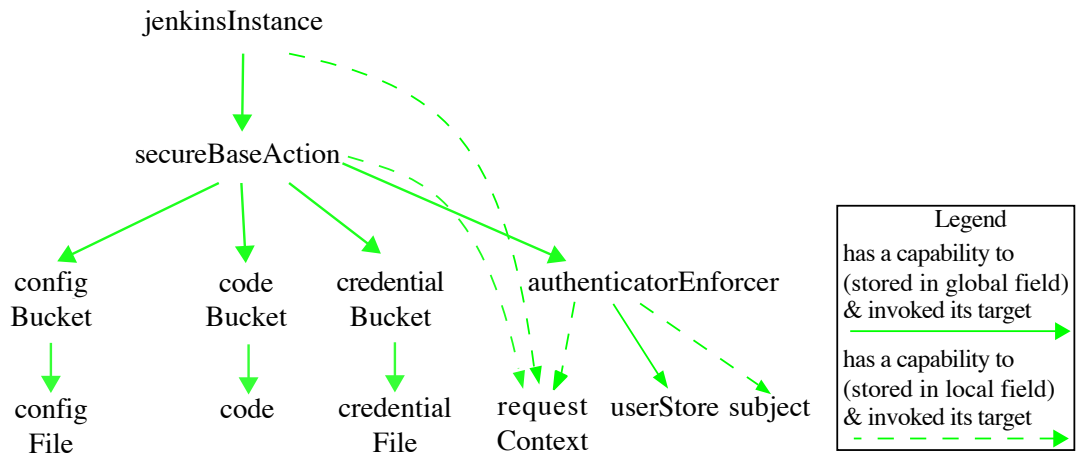


Figure 7.6: Jenkins with the Authenticator Enforcer Pattern. The *secureBaseAction* intermediates and authenticates *jenkinsInstance*'s access to *configBucket*, *codeBucket*, and *credentialBucket*. The *jenkinsInstance* has to store its identification information in *requestContext* and then provides the *requestContext* to *secureBaseAction*.

Listing 7.3: Verification Procedure of the Bucket (modified)

```
1 bktBreached(?Src,?T):-
2     !MATCH(?Src, ?T),
3     !MATCH(?Src,<Jenkins>),
4     !MATCH(?Src,<secureBaseAction>),
5     hasRef(?Src,?T).
```

In order to ensure that the build artifact is packaged into an image correctly, Jenkins is relieved from this duty and a trusted image builder is used. Furthermore, we want to be able to detect whether or not the image is tampered with during testing. We need an *imageBuilder* that will build the image, request an integrity check calculation (checksum) from *integrityChecker* and store the checksum in a database, *checksumStore*. This needs to be protected and which is thus classified as a secret. In order to achieve this, we create *imageBuilder* and connect Jenkins to it. Then, *imageBuilder* is connected to *integrityChecker* and *checksumStore*. As *checksumStore* is a secret and *imageBuilder* is trusted to have access to it, *imageBuilder* is added to the granted component set of

checksumStore using the grant tactic. Thus, a new Datalog rule is written (shown in Listing 7.4) to verify that no other component, except those in its granted component set (i.e. *imageBuilder*), can have access to *checksumStore*.

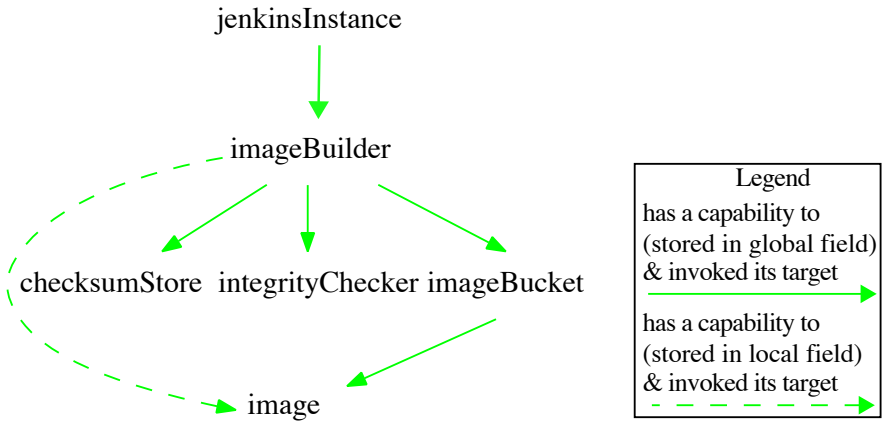


Figure 7.7: Jenkins with Image Builder and Integrity Checker. All the arrows are in green, which means that the functions of each component has been invoked.

Listing 7.4: Verification Procedure of the Checksum store

```
1 cSumBreached(?Src,?T):-
2     !MATCH(?Src, ?T),
3     !MATCH(?Src,<imageBuilder>),
4     hasRef(?Src,?T).
5 assert!cSumBreached(?Src,<checksumStore>).
```

In order to satisfy the claim that the correct version of the built image is deployed for testing and is not tampered with, we need to satisfy the claim that the exact path to images remains confidential. This is to ensure that the correct version of the image is deployed. We can consider two ways of protecting confidentiality of data: encryption and obfuscation. The encrypted storage pattern (Kienzle and Elder, 2002) aims to harden the confidentiality of a system. It encrypts data before storing it, and the encryption key must be stored securely. This mitigates the impact of the loss of a file to an

attacker, because the content of the file remains confidential, as it has been encrypted. Figure 7.8 shows the capability-specific design fragment of the encrypted storage pattern. The *encryptedStorage* component has access to the *storage*, *encryptedDecrypt* and *key* components. The user (invoker) sends an encrypt command, together with the data, to *encryptedStorage*. *encryptedStorage* loads the value of the key into *encryptDecrypt* and sends an encryptData command, together with the data, to it. The encrypted data is then returned to the *encryptedStorage*, which sends it back to the invoker, and it is stored in *storage*. The security property that is of interest is the confidentiality of the data. Since the data is encrypted, access to the encryption key needs to be minimized. Thus, only *encryptedStorage* is given access to the key. The security property is reflected in Listing 7.5. It checks whether any component in the design fragment, with the exception of *encryptedStorage*, has access to the *key*.

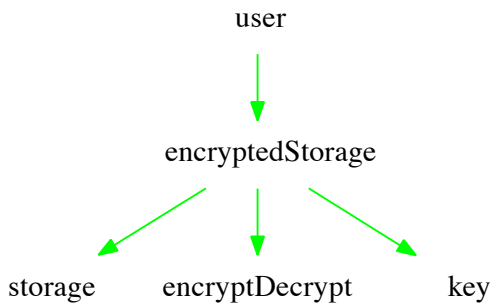


Figure 7.8: Encrypted Storage Design Fragment. All the arrows are in green, which means that the functions of each component has been invoked.

The existing model of the pipeline is composed with the encrypted storage design fragment by connecting *deployer* to *encryptedStorage* with the connect tactic. The *deployer* will generate a time-bound temporary URL of the image location in a bucket, a mechanism provided by AWS S3, and then invoke the encryptData function of *encryptedStorage* to encrypt the temporary URL. The temporary URL not only obfuscates the real location of the image, but also puts a time-bound to limit access to the image.

Listing 7.5: Verification Procedure of the Encrypted Storage Design Fragment

```
1 keyBreached(?Src,?T) :-
2     !MATCH(?Src, ?T),
3     !MATCH(?Src,<encryptedStorage>),
4     hasRef(?Src,?T).
5 assert !keyBreached(?Src,<key>).
```

However, we can only model the invocations and cannot check this property in SAM. The deployer then invokes AWS OpsWorks, which instructs the *ec2instance* to pull the image from the encrypted URL and then run that image on the instance. As the *storage* componentis not used, we disconnect *storage* from *encryptedStorage* and delete the storage, using the disconnect and delete tactics respectively. Once the test is completed, the release manager will be notified and has to make a decision whether or not the latest build should be deployed to production. The release manager will perform an integrity check on the latest build image to ensure that the image is untampered with. The integrity check can be done by calculating the checksum for the build image and comparing it with the entry in the *checksumStore*. If it matches, he can approve the latest build image to be deployed to the production environment through a different deployer. Having different deployers for the testing and production environments help ensure that there is no invocation between these two environments. One assumption is that the EC2 instances have been already been launched by AWS OpsWorks. Figure 7.9 shows the final model of the testing environment of the pipeline.

Finally, we want to ensure that there is no direct communication between the testing and production environments. The execution domain pattern (Schumacher et al., 2006) aims to restrict a process to specific resources by defining logical execution environments (domains). We define three different domains, which are testing, production and shared. The shared domain consists of utility components that are used by components in both

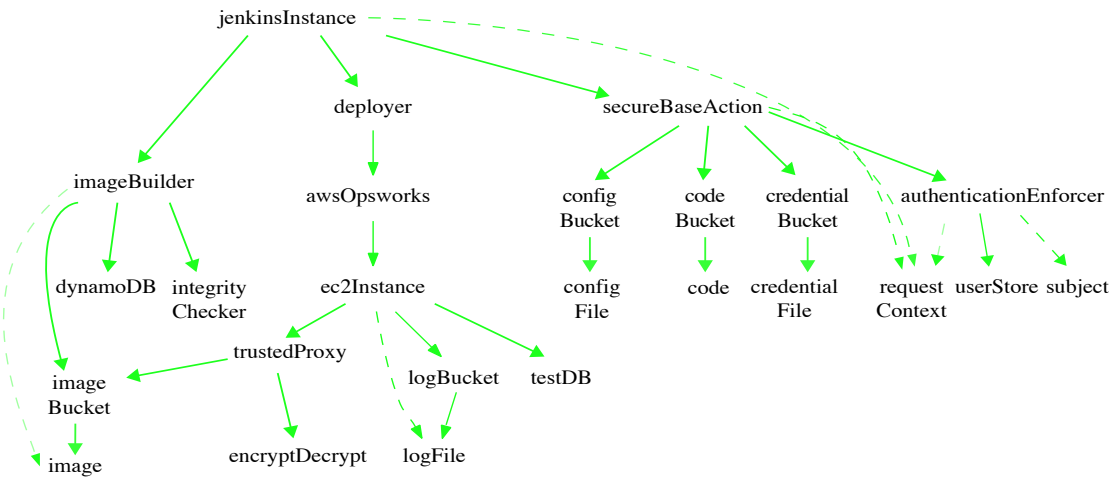


Figure 7.9: The testing environment of the Continuous Deployment (CD) pipeline. All the arrows are in green, which means that the functions of each component has been invoked.

production and testing domains. For specifying a component belongs to a particular domain, we use `hasIdentity(<component>,"domain name")`. To ensure that no component in the testing domain has access to the production domain, a Datalog rule, Listing 7.6, is written to check this property. The Datalog rule specify that there is a security flaw if a component in the testing environment has access to a component in the production environment.

Listing 7.6: Verification Procedure for no cross domain access

```
1 haveBadAcess(?Src,?T):-
2   hasRef(?Src,?T),
3   hasIdentity(?Src, "Testing"),
4   hasIdentity(?T, "Production").
```

In order to demonstrate this property, we connect *ec2Instance* from the testing environment to *ProductionDB* in the production environment. Figure 7.10 shows that there is a red arrow, which signifies a security violation, from *ec2Instance* to *ProductionDB*. This connection is considered as a bad access and is caught by the Datalog rule specified

above (Listing 7.6).

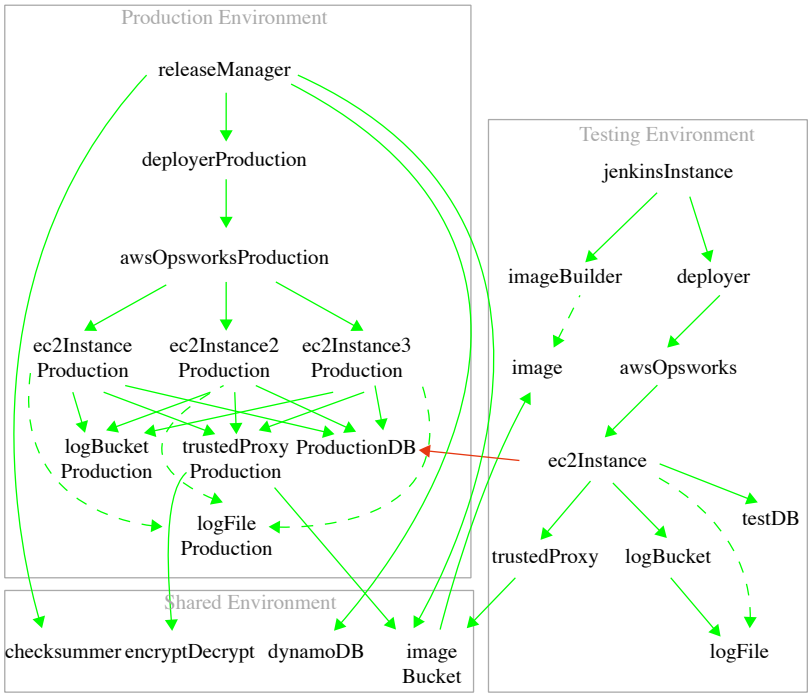


Figure 7.10: Testing to Production disallowed. There is a security violation, which is represented as one red arrow from the *ec2Instasnce* to the *ProductionDB*. This is a security violation because any components in the testing environment should not have access to a component in the production environment. The *ec2Instance* is a component in the testing environment while the *ProductionDB* is in the production environment.

The final pipeline design satisfies its security properties even though Jenkins and all the buckets and ec2instances are untrusted. Each verification procedure of the pipeline provides evidence to support the subclaims in the assurance case that is presented in Figure 7.2.

7.1.5 Discussion

The pipeline case study shows that the proposed approach can secure a CD pipeline in the presence of threats, which are defined in the threat model in Section 7.1.3. The pipeline is secured by gradually composing capability-specific design fragments with the existing

system design to eliminate the existing threats. This composition is done by using the composition primitives, which are defined in Section 5.1.

The case study also indicates that the composition primitives are sufficient to express different compositions that are required to secure the pipeline. The primitive tactics have been exercised and utilised multiple times in the case study. Complex security-critical systems can be built using combinations of these primitive tactics. However, due to the tactics being very primitive, it might be tedious to perform commonly used combinations, such as the proxy tactic, which I define in Section 5.3. The proxy tactic is a higher-level tactic that helps to insert a new component in between two connected components. This can be achieved through multiple application of connect and disconnect tactics. Building a catalog of these higher-level tactics is crucial as it may ease the composition approach.

When I use a capability-specific design fragment for composition, I also reuse its verification procedure to help verify the security properties of the application design. Each design fragment has been verified individually before the composition to identify and remove localized problems to an individual design fragment before those problems propagate to the whole application design through composition. This is intended to reduce the verification effort and design effort.

7.2 Smart Meter

In this section, I evaluate the applicability of my approach and the higher level tactics that are defined using the composition primitives using a Smart Meter with requirements derived from industrial standards. These higher level tactics are derived from multiple applications of the primitives that are introduced in Section 5.1.

Section 7.2.1 introduces the concept of a smart meter, which is part of the Advanced

Metering Infrastructure (AMI), and provides background, including an overview of the requirements and architecture. Section 7.2.2 describes the application of my process in designing a secure smart meter before concluding this chapter with discussion in Section 7.2.3.

7.2.1 Background

A smart meter is an electronic device that records energy usage and supports two-way communication with utility providers. It transmits information back to utility providers and receives information from utility providers. A smart meter is installed in the home of an end user and is part of the Advanced Metering Infrastructure (AMI). Figure 7.12 shows the architecture design of the AMI, in which a meter has to communicate with AMI Head End, a display device, AMI communications network device and field tool/device.

I use the Advanced Metering Infrastructure (AMI) (The Advanced Security Acceleration Project, 2010) is used as a context for a realistic smart meter example. The AMI Security Profile (The Advanced Security Acceleration Project, 2010) has 130 security requirements, based on NIST 800-53 (Joint Task Force Transformation Initiative, 2010), a US Federal document defining the security and privacy controls for information systems and organizations. The AMI is large, so the focus is on a particular component, the smart meter. A smart meter is an electricity meter capable of two-way communications with a utility company. From the use cases provided by Smartgridipedia[10] and security requirements from the security profile, a model of a smart meter supporting the functionality and the security features of the meter is derived. The operational (e.g. update regularly) and organizational (e.g. personnel training) requirements are filtered out and the requirements concerning all components in the AMI and those specific to the meter

[10]http://www.smartgridipedia.org

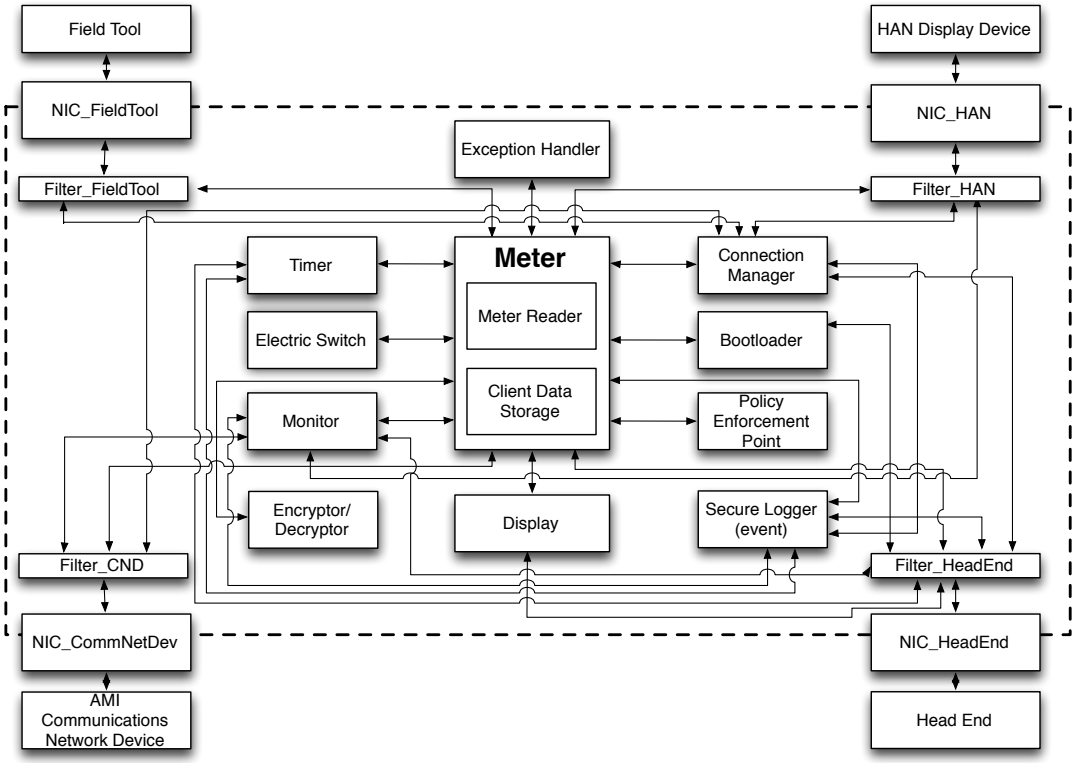


Figure 7.11: Smart Meter Architecture Full Perspective

are concentrated on. Based on these criteria, there are 53 relevant security requirements. All the relevant security requirements are listed in Appendix A. From these relevant security requirements and use cases, a smart meter architecture design is devised as shown in Figure 7.11.

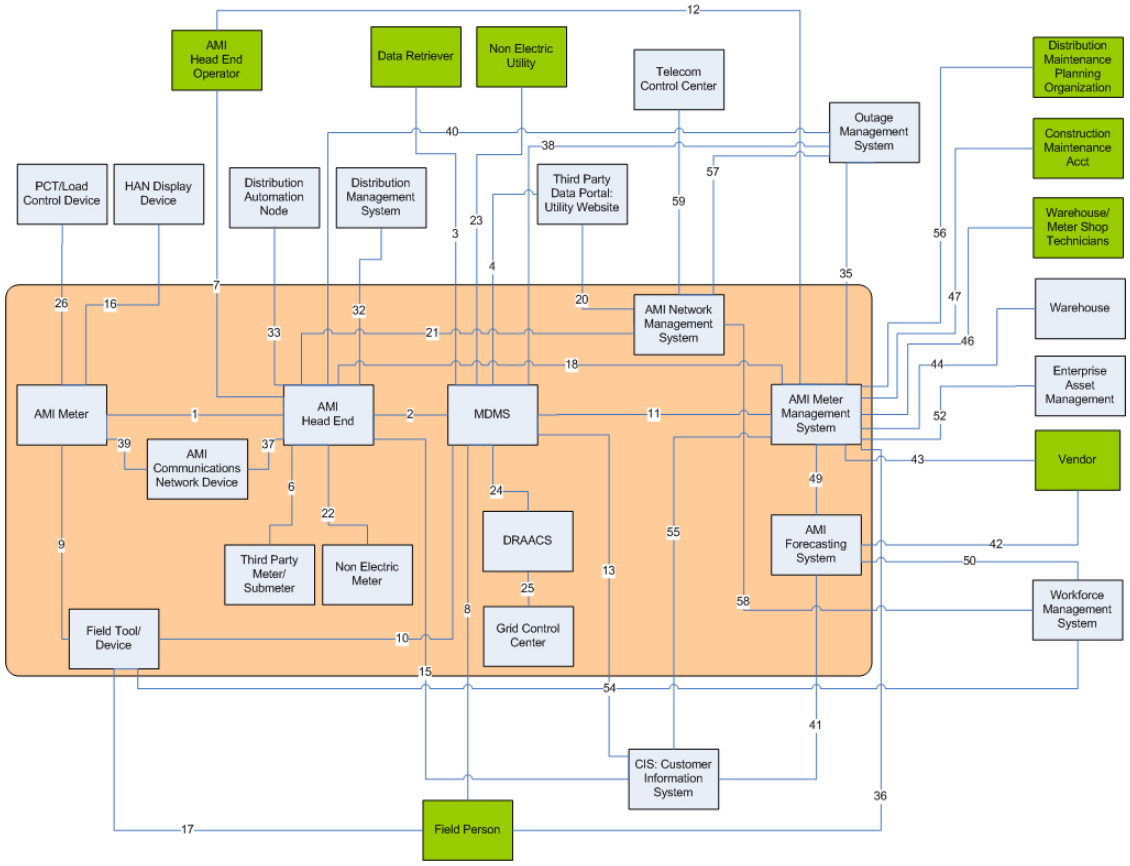


Figure 7.12: AMI Architecture Full Perspective (The Advanced Security Acceleration Project, 2010)

7.2.2 Secure Smart Meter Design

Here, I focus on the design and evaluation of components implementing the logging requirements. The logger is the component with the most connections to other components. This is due to a security requirement, DHS-2.14.4, which specifies that all components in the AMI shall log all security events.

Table 7.2: Smart Meter Logging Security Requirements

Requirement	Requirement Detail
DHS-2.14.4	Components of AMI shall log all security events
DHS-2.15.20	Unsuccessful login attempts shall be logged
DHS-2.16.2	AMI components shall generate log files
DHS-2.16.9	Protect audit information (log files) from unauthorized access

Table 7.2 shows relevant logging-related requirements. DHS-2.16.2 requires the creation of a log file, while DHS-2.14.4 and DHS-2.15.20 describe some of the log contents. Requirement DHS-2.16.9 states that the log file has to be protected from unauthorized access. Based on this requirement, the security property that is of interest is the confidentiality of the contents of the log file. Covert channels and other access properties are not dealt with in this work.

The assurance case for the logging mechanism in Figure 7.13. It is built with the ASCE tool[11], using the Claim-Argument-Evidence (CAE) notation (Bloomfield and Bishop, 2010). Claims represent what must be demonstrated by the system. An argument provides reasoning as to why the claim has been met by the supporting evidence. In Figure 7.13, the blue oval shape represents the claims and subclaims. The green rounded rectangles and pink rectangles represent the arguments and evidence respectively. The black hexagons represent assumptions. The requirement DHS-2.16.9 is set as the top-level claim. Subclaims are then identified to support the top-level claim. These subclaims

[11] ASCE — http://www.adelard.com/asce/

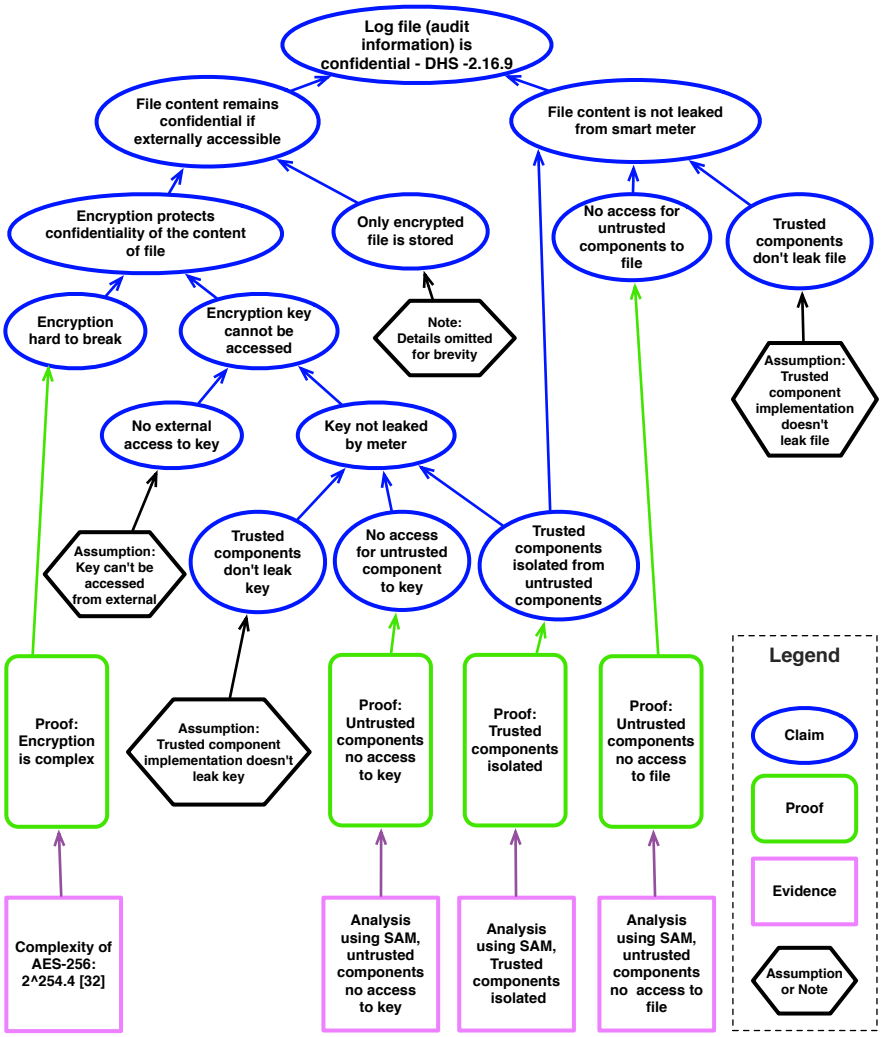


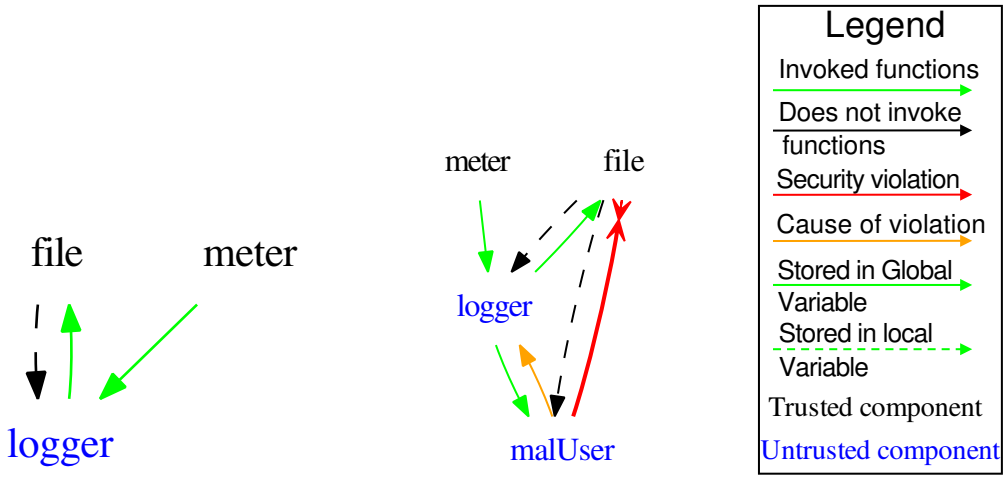
Figure 7.13: Assurance Case for Smart Meter Logging

demonstrate that the meter remains secure despite possible attacks. Arguments and evidence for the claim are then provided. During the design process, the assurance case is incrementally evolve as the design changes to defeat new attacks.

The log file can be accessed either from within the device or external to the device. External access is direct access to the storage without the controls provided by the system design. Internal access is within the device, i.e. behind the interfaces to the device. We claim that the contents of the file will remain confidential even if it is externally accessible. This is supported by subclaims about the encryption that protects the confidentiality

of the log file. One assumption is that the key cannot be accessed externally (e.g. by being stored in a tamper-resistant store). Each of these subclaims is supported by another subclaim or an argument and associated evidence. For instance, we argue that the key is not leaked by the meter, and support this by three other subclaims. One of them is that untrusted components do not have access to the key. An untrusted component might potentially communicate the contents of the key if it has access to it. Therefore, it is essential to ensure that only trusted components have access to the key. An analysis in SAM that only trusted components have access to the encryption key is provided as a piece of evidence. Furthermore, the fact that encryption is hard to break, e.g. as Bogdanov et al. (2011) have shown, the computational complexity of a full key recovery attack on AES-256 is $2^{254.4}$, is added as a subclaim of the "encrypted protection" claim. A claim can be used as subclaim to more than one claim (or subclaim). For instance, the claim "trusted components are isolated from untrusted components" is a subclaim to both the "file content is not leaked from smart meter" and "key is not leaked by the meter" claims.

We start by building a model that satisfies the requirements for the need of logging (DHS-2.14.4 and DHS-2.15.20). This model is referred to as Model I and is shown in Figure 7.14a. However, this model does not sufficiently address the requirement to protect log files from unauthorized access (DHS-2.16.9). This is shown by introducing an attack from a malicious user assumed to have access to the logger. As both the malicious user and logger are untrusted (blue in SAM notation), they will pass and take any capability that they are able to. The logger grants file access to itself. The malicious user (malUser) grants the logger and the file access to itself. The security violation is caused by the malicious user taking the loggers capability to the file. The model resulting from the analysis is shown in Figure 7.14b.A Datalog query `hasRef(<malUser>,<file>)` is defined to check whether there is a reference from



(a) Baseline. In this figure, the *meter* has an access to *logger* and invoked its functions. The *logger* has an access to *file* and has invoked its functions. The *file* gained an access to *logger* but does not invoke any functions of *logger*

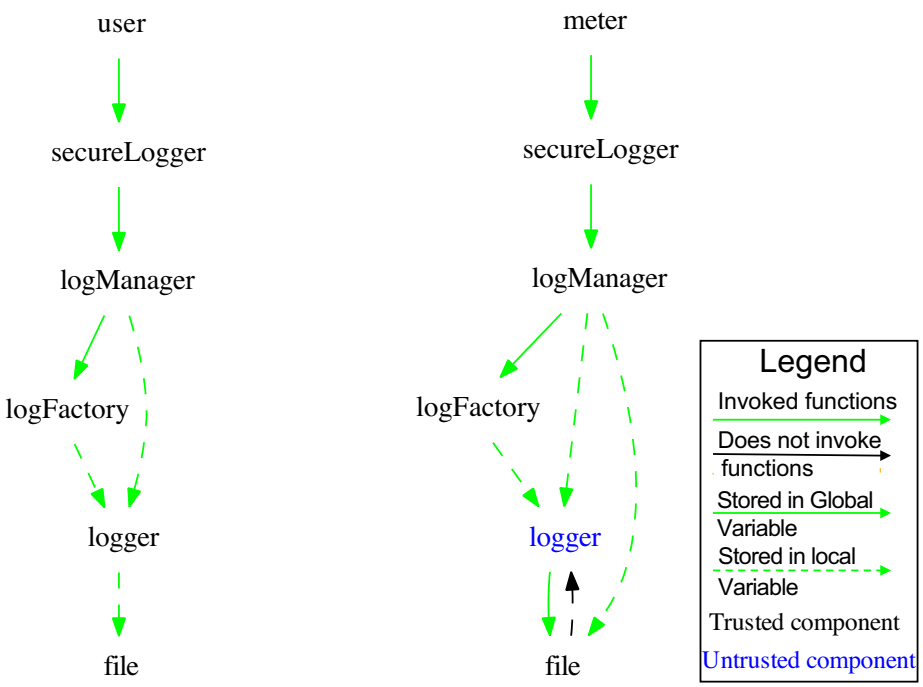
(b) Model I with malUser Attack. There is a security violation in this figure, which is represented as a red arrow from the *malUser* to the *file*. The cause of this violation is represented as an orange arrow from the *malUser* to the *logger*. The *malUser* should not have an access to the *file* but gained it from the *logger*

Figure 7.14: Model I

malicious user to file. The result from the query shows that the design of Model I has not satisfied the requirement DHS-2.16.9. The red arrow from malUser to file in Figure 7.14b indicates that the malicious user has access to the file, which is a security violation.

One of the subclaims of the assurance case is that there is no access for untrusted components to the file. We use the secure logger design fragment that has been shown in Section 4.5 to mitigate this attack. One of the goals of the secure logger pattern is to decouple the logging functionality from the application, so that only authorized users are able to view the contents of the log file, as shown in Figure 7.15a. We aim to use the secure logger design fragments verified property that only authorized users can access the file.

In this design fragment, a user sends a log command to *secureLogger* with data. Upon receipt, the *secureLogger*, whose main responsibility is to collect the data, sends the data with a log command to the *logManager*. The *logManager* will request a new



(a) Secure Logger design fragment. All the arrows are in green, which means that the functions of each component has been invoked.

(b) Model II baseline. All the arroows, except *file* to *logger*, are in green. The arrow from *file* to *logger* is black, which means that *file* does not invoke the functions of *logger*.

Figure 7.15: Secure Logger Design Fragment and Model II

instance of logger from logFactory. The logger is the component that logs the data. As shown in Figure 7.15a, the *secureLogger* has a capability to *logManager*, and the *logManager* has a capability to both the *logFactory* and the newly created *logger*. We verify that the user does not have direct access to the file. A Datalog rule is written for the Points-To analysis engine built into SAM to analyze this fragment. This rule defines whether there is an access from *user* to *file*. We check that the user does not have access to the file with the query `!hasRef(<user>,<file>)`.

The secure logger design fragment is applied to Model I, and the resulting model is called Model II (with secure logger). Figure 7.15b depicts Model II in SAM. Reusing the Datalog rule defined for the secure logger design fragment, `hasRef(<meter>, <file>)`, we can check that the meter does not have access to the file. This step is essential to ensure that composing with the secure logger pattern does not break the

property previously achieved, which is that meter does not have access to file.

The malicious user attack that broke Model I is then reintroduced. Figure 7.16 depicts Model II after the introduction of the malicious user assumed to have access to the *secureLogger*. The malicious user has access to *secureLogger*, *loggerUnknown* and *fileUnknown* in the figure. The *loggerUnknown* and *fileUnknown* components are a logger instance and a log file dynamically created for the malicious user respectively. Using the Datalog query `!hasRef(<malUser>,<file>)`, we confirm that the malicious user does not have access to the log file. Therefore, the secure logger design fragments has mitigated that attack and satisfies the claim that untrusted components do not have access to file. An argument supporting this claim is added to the assurance case with the verification as supporting evidence.

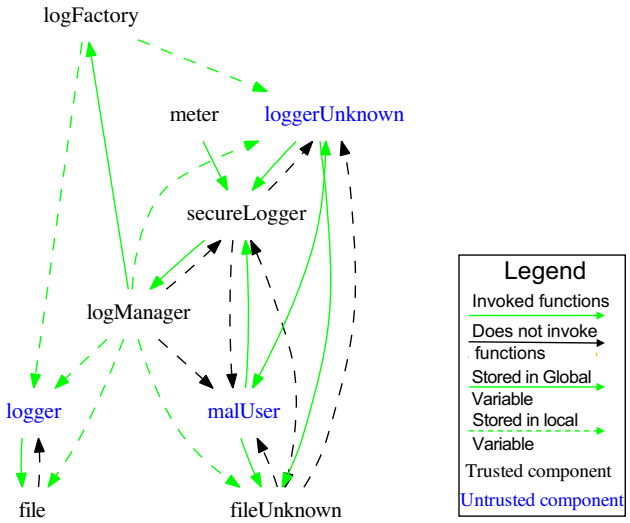


Figure 7.16: Model II with Malicious User Attack. This figure shows that the *malUser* does not have access to *file*. The *logger*, *loggerUnknown* and *malUser* components are components with untrusted behaviors.

One major subclaim in our assurance case is that file remains confidential even if there is external access to it. To demonstrate that this subclaim is valid, the *malUser* is granted a capability to the log file directly. This can be achieved in SAM by making the log file public, which means that all untrusted components are granted access to it. This

attack is introduced to Model II and analyze whether the confidentiality property of the model stays intact. As seen in Figure 7.17, the malicious user has access to the file and thus the assertion for the Datalog query `!hasRef(<malUser>,<file>)` fails. So, the information inside the log file can be read by the malicious user. Actions must be taken to mitigate this attack and harden the system.

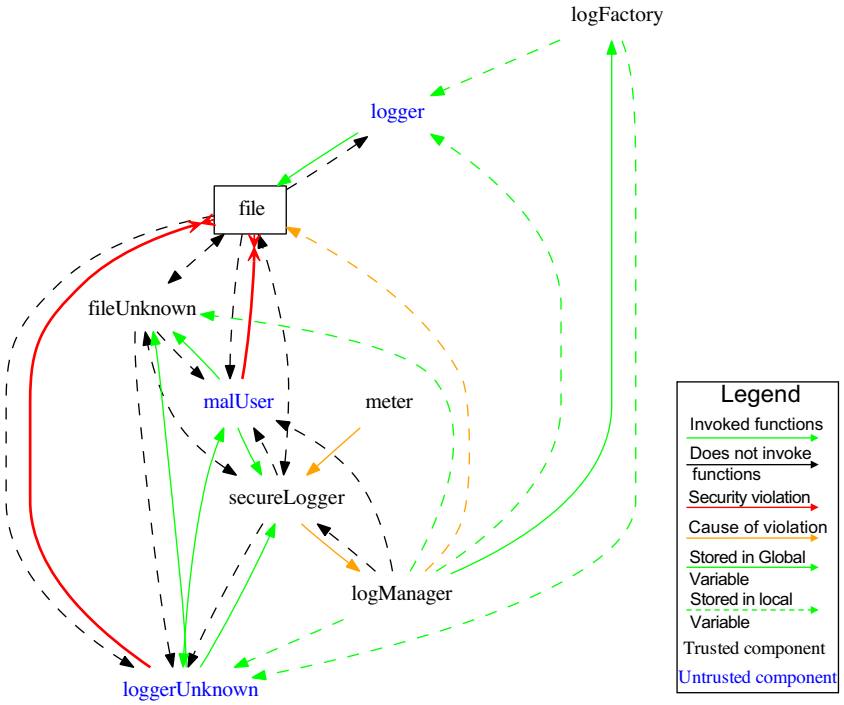


Figure 7.17: Model II with External Access Attack. There are two security violations, which are represented as two red arrows: *malUser* to *file* and *loggerUnknown* to *file*. The box around the *file* component depicts that it is made public, which means that all untrusted components are granted access to it.

The encrypted storage pattern (Kienzle and Elder, 2002) aims to harden the confidentiality of a system. It encrypts data before storing it, and the encryption key must be stored securely. This mitigates the impact of the loss of a file to an attacker, because the content of the file remains confidential, as it has been encrypted. Figure 7.18 shows the capability-specific design fragments of the encrypted storage pattern. The *encryptedStorage* component has access to storage, *encryptor_decryptor* and key components. The user sends a write command, together with the data, to *encryptedStorage*.

encryptedStorage loads the value of the key to the *encryptor_decryptor* and sends an encryptData command, together with the data, to it. The encrypted data is then returned to *encryptedStorage*, which will send the encrypted data to be stored by storage. The security property that is of interest is the confidentiality of the data. Since the data is encrypted, access to the encryption key needs to be minimized. Thus, only *encrypted-Storage* is given access to the key. A new Datalog rule, `isSecBreached`, is defined as shown in Listing 7.7, to verify this property.

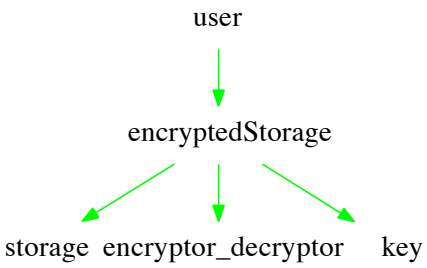


Figure 7.18: Encrypted Storage Design Fragment. All the arrows are in green, which means that the functions of each component has been invoked.

Listing 7.7: Encrypted Storage Verification Procedure

```
1 declare isSecBreached(Ref Source, Ref Target).
2 isSecBreached(?Source,?Target) :-
3     !MATCH(?Source, <encryptedStorage>),
4     !MATCH(?Source,?Target),
5     hasRef(?Source,?Target)
6 assert !isSecBreached(?Source,<key>).
```

The rule specifies that the security is breached if there is a component in the model, other than *encryptedStorage*, that has access to the key and storage at the same time. The design fragment model is checked using the query `!isSecBreached(?Source,<key>,<storage>)`. When the model satisfies the rule, we compose it together with Model II and call it Model III. It is the composite

model consisting of the Model I, secure logger design fragment and encrypted storage design fragment.

For the composition, we use the connect and replace tactics defined in Chapter 5. The encrypted storage design fragment is connected to the *secureLogger* component of the secure logger pattern. Then, the *storage* component of the encrypted storage pattern is replaced with the *logManager* component. Figure 7.19 shows Model III (with secure logger and encrypted storage design fragments). The *secureLogger* component has a capability to the *encryptedStorage* instead of the *logManager*. The *encryptedStorage* will encrypt, which is done by the *encryptor_Decryptor*, the data that are received from the *secureLogger* and will send it with the log command to the *logManager*. The *logManager* will request a new instance of *logger* from the *logFactory* and will command the *logger* to log the encrypted data.

Listing 7.8: Verification Procedure of Model III

```
1 declare isSecBreached(Ref Source, Ref Target, Ref T2).
2 isSecBreached(?Source,?Target, ?T2) :-
3    !MATCH(?Source, ?Target),
4    !MATCH(?Source, ?T2),
5    hasRef(?Source,?Target),
6    hasRef(?Source,?T2).
7 assert !isSecBreached(?Source,<key>,<file>).
8 assert !hasRef(<meter>,<file>).
```

First, this model is analyzed with the malicious user attack to make sure that it does not break the properties that Model II already has. We run the same analysis as for Model II and update the Datalog rule, `isSecBreached`, for Model III (with secure logger and encrypted storage). The verification procedure (line 7 in Listing 7.8) from the secure logger design fragments is reused to verify Model III. The Datalog rule,

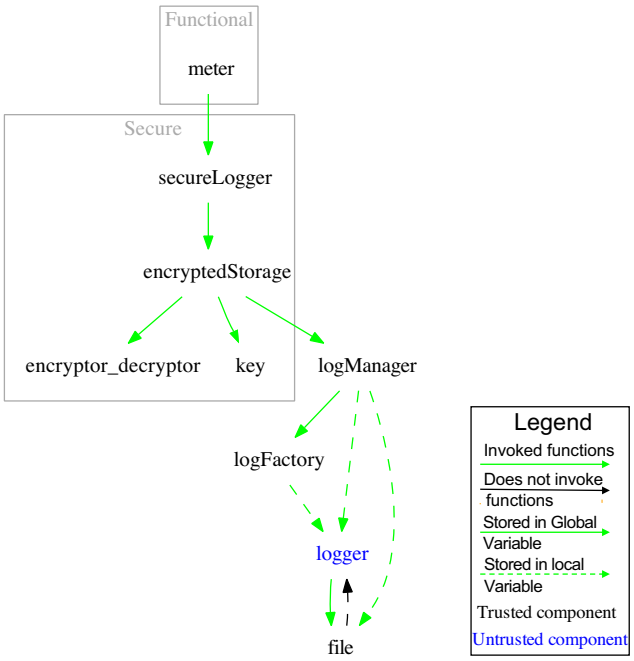


Figure 7.19: Model III (with Secure Logger and Encrypted Storage). All the arrows are in green, except the black arrow *file* to *logger*, which means that the functions of each component has been invoked. The black arrow depicts that *file* do not invoke any function of *logger*.

defined for encrypted storage fragments, is reused to verify Model III. The rule for Model III is updated to state that security is breached if there is a component, other than *encryptedStorage*, that has access to the key and file at the same time. This is because the content of the file has been encrypted and the key is required to decrypt the file. The rule is updated by modifying the declaration of the rule and update the rule, as shown in lines 1-6 of Listing 7.8. We then check the rule with a query in SAM, `!isSecBreached(?Source,<key>,<file>)`, to show that the confidentiality of the log file remains intact despite of the introduction of the malicious user attack. Figure 7.20 shows Model III with the malicious user attack, in which the malicious user does not have access to file.

In Figure 7.21, the malicious user is given direct access to the file. Recall that the contents of the file have been encrypted and are incomprehensible without decryption,

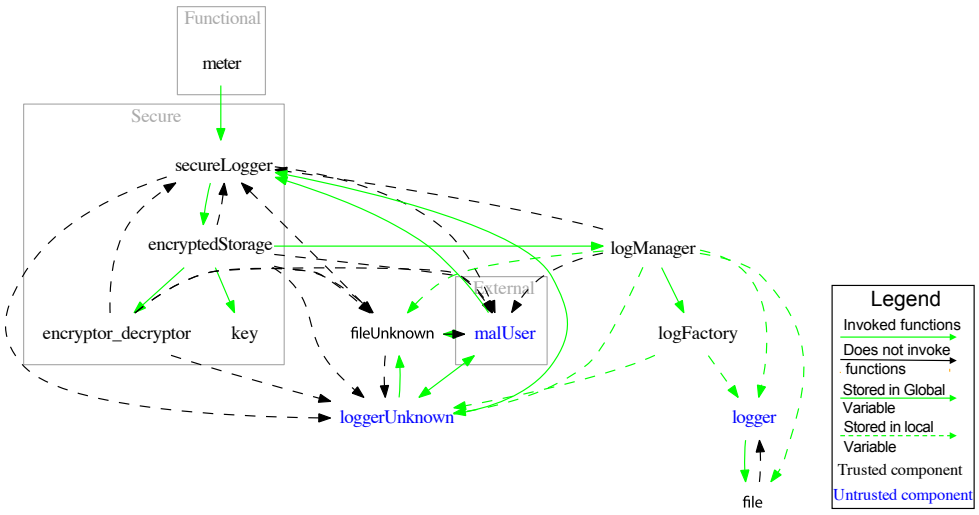


Figure 7.20: Model III with Malicious User Attack. This figure shows the state of Model III after all the possible accesses have been propagated in the presence of the *malUser*. There is no component, except the *encryptedStorage*, that has an access to both the *file* and *key* simultaneously.

which requires the encryption key. We have shown, by checking the Datalog query `!hasRef(<meter>,<file>)`, that the malicious user does not have any access to the key. As a result, the confidentiality property of the model is still intact. This argument is added to the assurance case with verification as supporting evidence.

Other sub claims have been identified as shown in Figure 7.13. These are that modification to the key does not compromise the file and that the encryption method used is hard to break. For the former, we claim that the confidentiality of the log file will remain intact in case the key is modified. We argue that the encrypted content of the file is incomprehensible without decryption. As the encryption key has been changed, there is no way to decrypt the file and the content of the file cannot be interpreted. Thus, the content of the file remains confidential. As for the latter, NIST 800-53 recommends using an existing algorithm rather than creating a new encryption algorithm. These are included in our assurance case as it gives an indication of the impact of changes to the system and its security properties.

Authentication and authorization are crucial components in a secure application.

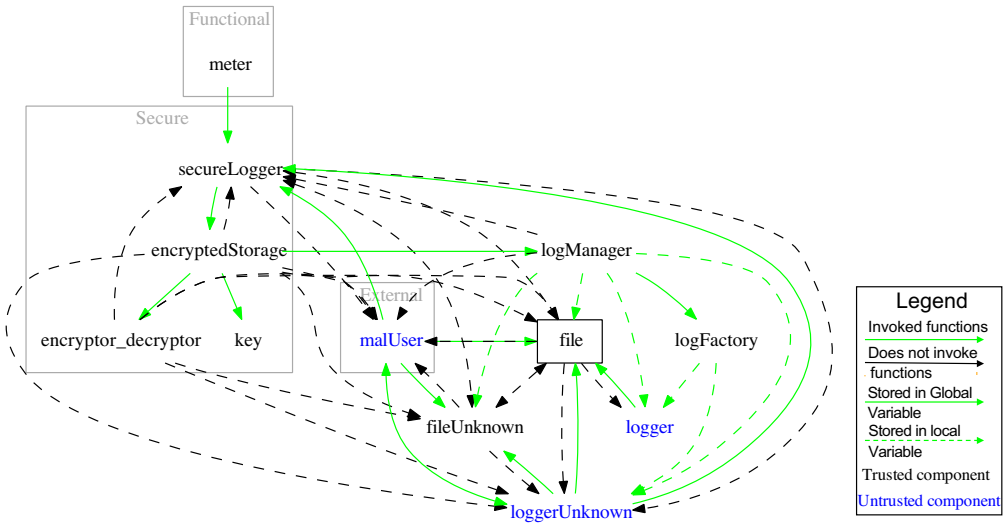


Figure 7.21: Model III with External Access Attack. This figure shows the state of Model III after all the possible accesses have been propagated in the presence of the *malUser* and the external access attack. There is no component, except the *encryptedStorage*, that has an access to both the *file* and *key* simultaneously. The box around the *file* component depicts that it is made public, which means that all untrusted components are granted access to it.

Table 7.3 shows the security requirements regarding authentication and authorization. Based on these requirements, Model III is further enhanced with the authentication enforcer (Schumacher et al., 2006) and authorization enforcer (Schumacher et al., 2006) patterns. These patterns help cancel out one of the assumptions of the secure logger pattern, which is the authentication and authorization mechanisms are present in the system.

In the authentication enforcer design fragment, a user creates a *requestContext* and sends it together with an authenticate command to the *authenticationEnforcer*. In this design fragment, a *requestContext* is an object that contains the user credentials required for authentication. Upon receipt, the *authenticationEnforcer* will retrieve the credentials from the *requestContext* and will then verify the credentials with those stored in the *userStore*. Upon successful verification, the *authenticationEnforcer* creates a *subject* for that user. The design fragment is depicted in Figure 7.22.

We need to ensure that access to the *userStore* is limited to only the *authenticationEn-*

Table 7.3: Authentication & Authorization Requirements

Requirement	Requirement Detail
DHS-2.14.8, DHS-2.14.9, DHS-2.15.19	The need for Authentication and Authorization
DHS-2.15.7	Enforce authorizations for controlling access to the system
DHS-2.15.10, DHS-2.15.12	Components shall identify and authenticate users/components
DHS-2.15.14	Employ authentication methods to a cryptographic module

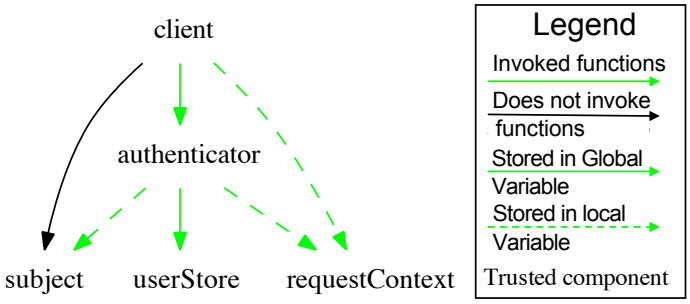


Figure 7.22: Authentication Enforcer Design Fragment

forcer. This is captured using the security property template defined in Section 6.3: `all no access to userStore, except authenticationEnforcer`. This is then translated into a Datalog rule, `authenticationBreached`, as shown in Listing 7.9. This Datalog rule checks whether any components in the system, except the *authenticationEnforcer*, has access to *userStore*. A security violation is detected if any components, except the *authenticationEnforcer*, has access to *userStore*.

In the authorization enforcer design fragment, a user creates a *requestContext* and sends it together with an authorize command to the *secureBaseAction*. *requestContext* is an object that stores information, such as user credentials and authentication token, required for authorization purposes. Upon receipt, *secureBaseAction* will retrieve the information stored in *requestContext* and will trigger the authorize command of *authorizationEnforcer*. *authorizationEnforcer* will call the authorize function of *authoriza-*

Listing 7.9: Verification Procedure of Authentication Enforcer

```
1 declare authenticationBreached(Ref Source,Ref Target).
2 authenticationBreached(?Source,?Target):-
3     !MATCH(?Source,<authenticationEnforcer>),
4     !MATCH(?Source,?Target),
5     hasRef(?Source,?Target).
6 assert !authenticationBreached(?Source,<userStore>).
```

tionProvider, passing the information as parameters, to get the rights associated with the user. *authorizationProvider* retrieves the rights associated with the user and creates *permissionCollection*. *permissionCollection*, which contains access rights of the user, is then returned to the user by *secureBaseAction*. The design fragment is shown in Figure 7.23.

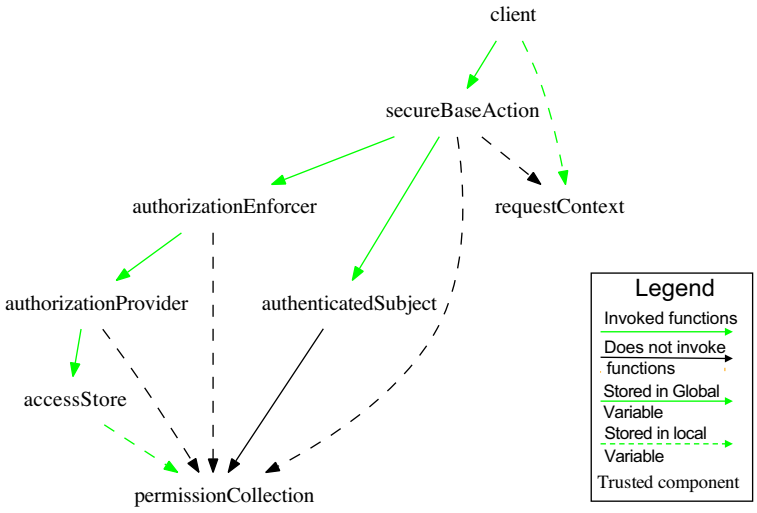


Figure 7.23: Authorization Enforcer Design Fragment

Figure 7.24 shows the resulting design and Listing 7.11 shows the corresponding verification procedures. We concatenate the verification procedures of Listing 7.8 with those of the authentication enforcer (lines 8-12) and authorization enforcer (lines 13-17) design fragments. Lines 8-12 specify that authentication is breached if there is a component in

Listing 7.10: Verification Procedure of Authorization Enforcer

```
1 declare authorizationBreached(Ref Source,Ref Target).
2 authorizationBreached(?Source,?Target):-
3     !MATCH(?Source,<authorizationProvider>),
4     !MATCH(?Source,?Target),
5     hasRef(?Source,?Target).
6 assert !authorizationBreached(?Source,<accessStore>).
```

the model, other than *authenticationEnforcer*, that has access to *userStore*. Lines 13-17 specify that the authorization is breached if there is a component in the model, other than *authorizationProvider*, that has access to the *accessStore*, which contains the collection of access rights for components. We need to ensure that access to the *accessStore* is limited to *authorizationProvider*. Thus, this property is captured in the template as such: all no access to accessStore, except authorizationProvider.

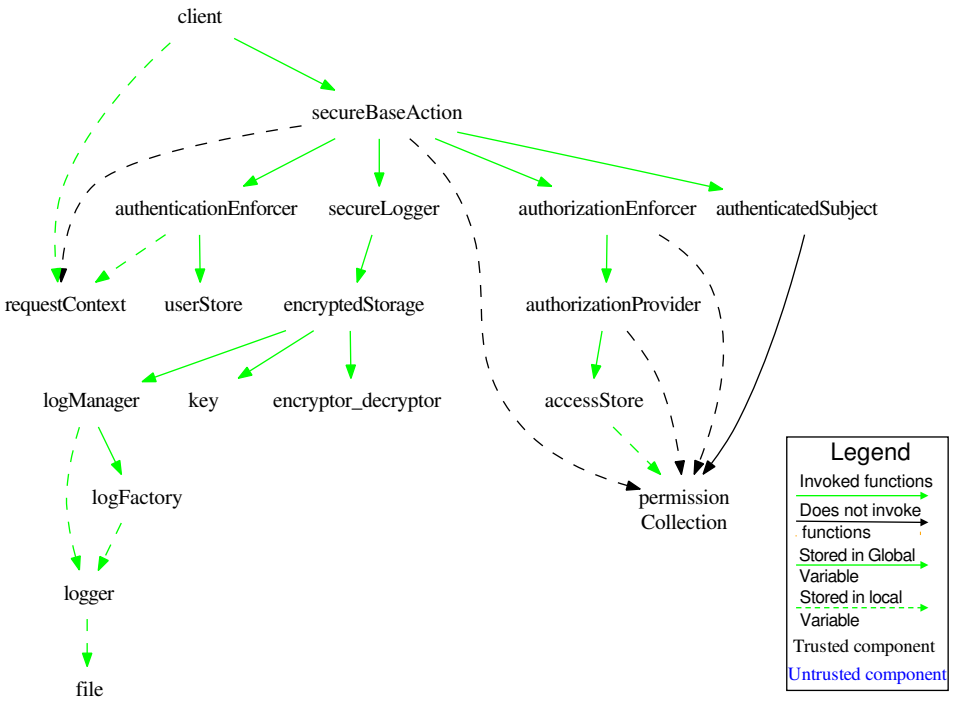


Figure 7.24: Model IV

Model IV with malicious user attack is shown in Figure 7.25 while Model IV with external access attack is shown in Figure 7.26. We analyze this model with the malicious user attack and external access attack. The confidentiality of the content of the log file is still intact and, on top of that, there is no unauthorized access to *accessStore* and *userStore* in the model during these attacks.

Listing 7.11: Model IV verification procedure

```
1 declare isSecBreached(Ref Source, Ref Target, Ref T2).
2 isSecBreached(?Source,?Target, ?T2) :-
3    !MATCH(?Source, <encryptedStorage>),
4    hasRef(?Source,?Target),
5    hasRef(?Source,?T2).
6 assert !isSecBreached(?Source,<file>,<key>).
7 assert !hasRef(<meter>,<file>).
8 declare authenticationBreached(Ref Source, Ref Target).
9 authenticationBreached(?Source,?Target) :-
10    !MATCH(?Source,<authenticationEnforcer>),
11    hasRef(?Source,?Target).
12 assert !authenticationBreached(?Source,<userStore>).
13 declare authorizationBreached(Ref Source, Ref T).
14 authorizationBreached(?Source,?T):-
15    !MATCH(?Source,<authorizationProvider>),
16    hasRef(?Source,?T).
17 assert !authorizationBreached(?Source,<accessStore>).
```

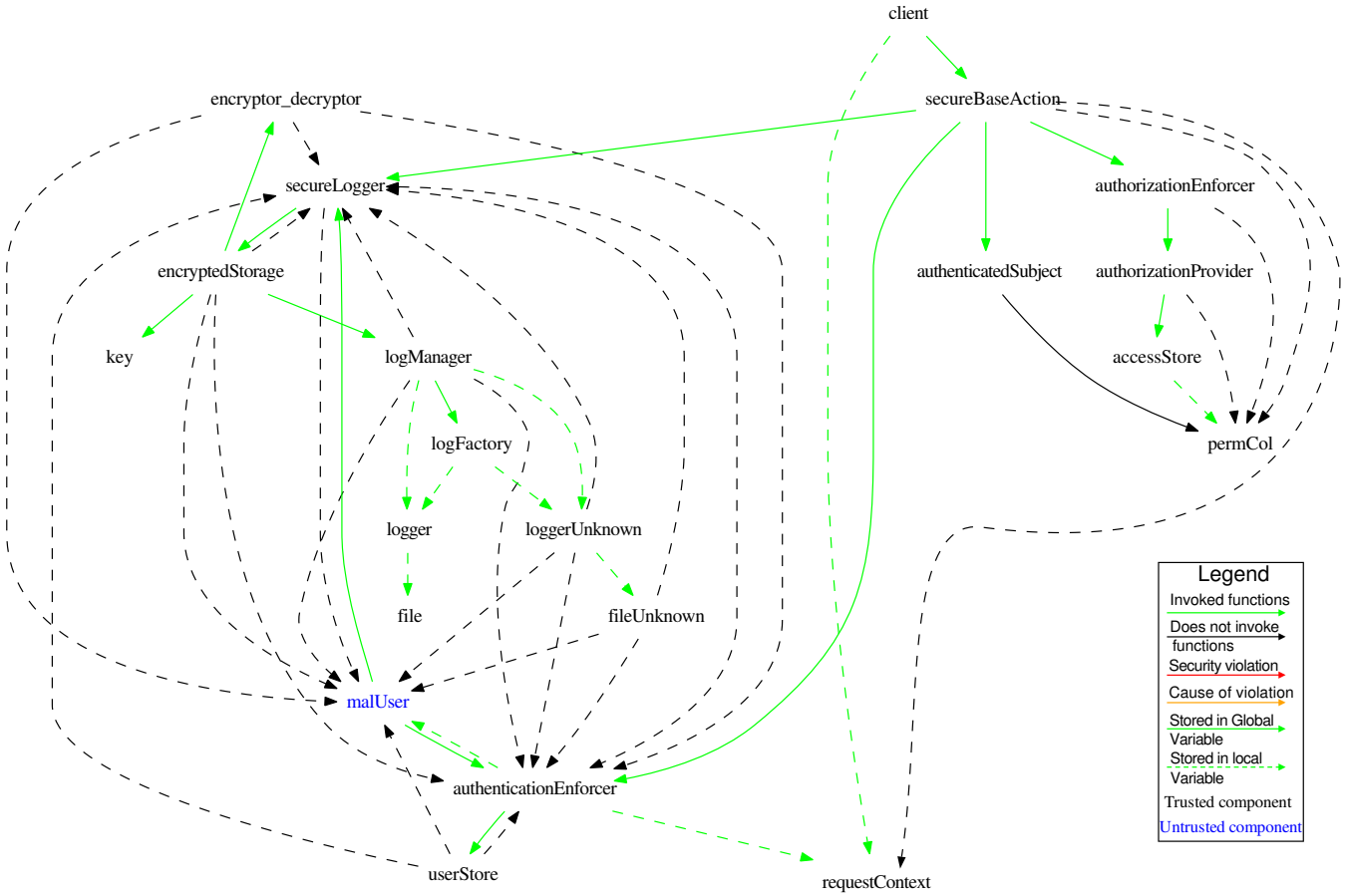


Figure 7.25: Model IV with Malicious User Attack. This figure shows the state of Model IV after all the possible accesses have been propagated in the presence of the *malUser*. There is no component, except the *encryptedStorage*, that has an access to both the *file* and *key* simultaneously. Furthermore, only the *authorizationProvider* that has an access to the *accessStore* component. In addition, there is no component, except the *authenticationEnforcer*, has an access to *userStore*. The box around the *file* component depicts that it is made public, which means that all untrusted components are granted access to it.

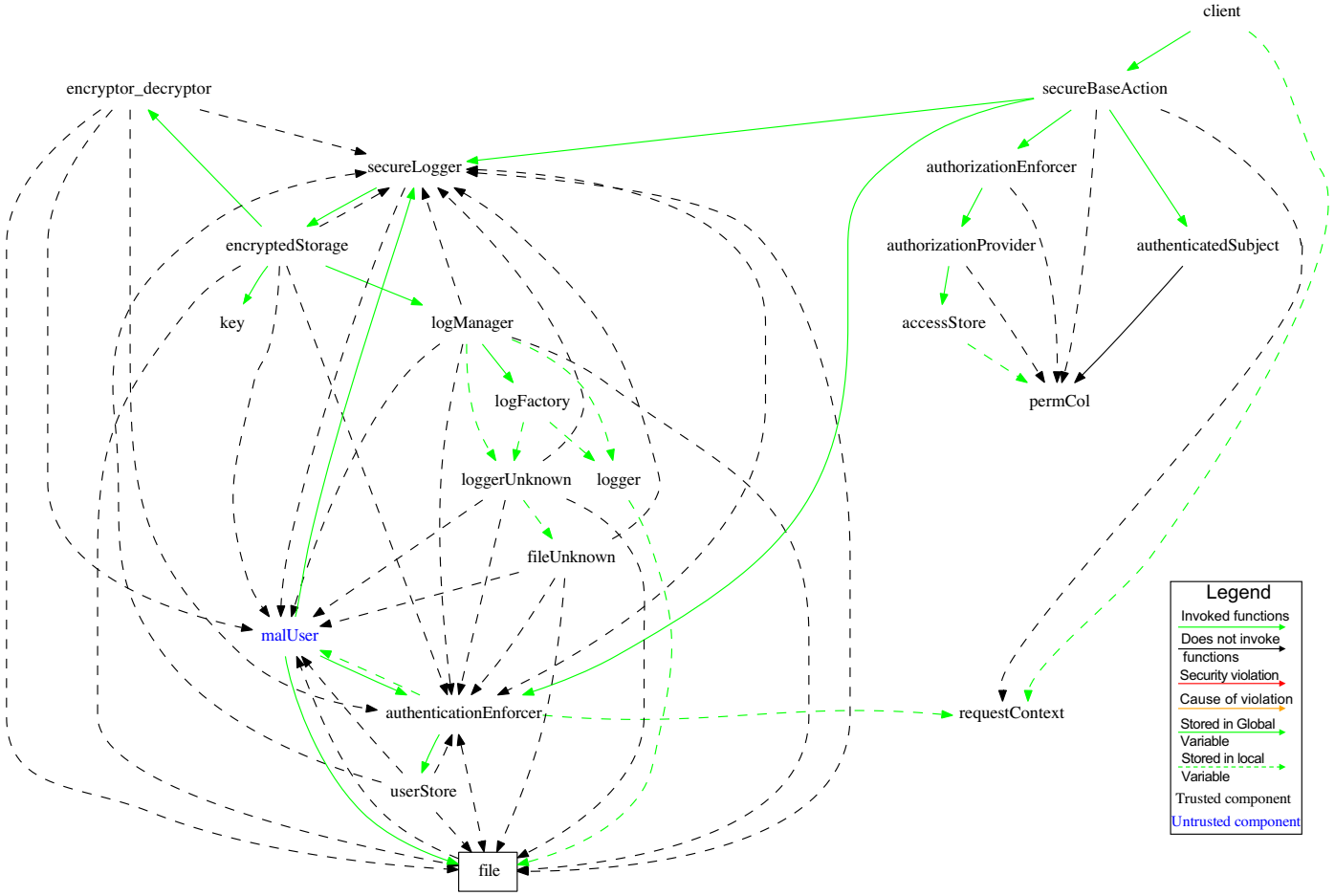


Figure 7.26: Model IV with External Access Attack. This figure shows the state of Model IV after all the possible accesses have been propagated in the presence of the *malUser* and the external access attack. There is no component, except the *encryptedStorage*, that has an access to both the *file* and *key* simultaneously. Furthermore, only the *authorizationProvider* that has an access to the *accessStore* component. In addition, there is no component, except the *authenticationEnforcer*, has an access to *userStore*. The box around the *file* component depicts that it is made public, which means that all untrusted components are granted access to it.

7.2.3 Discussion

The smart meter case study shows that the proposed approach can help to design a secure smart meter and that the higher level tactics defined in Section 5.3 are feasible to be applied. I compose an existing simple meter design with several design fragments, using the composition tactics defined in Chapter 5, to mitigate threats and harden the smart meter. Each design fragment has been verified individually before the composition to identify and remove localized problems to an individual design fragment before those problems propagate to the whole application design through composition. The verification procedure of each the design fragments that is used during composition is reused to verify the resulting application design. Reusing capability-based verified design fragments reduce the efforts required to not only build a secure design but also verify the resulting design.

The case study also indicates that the composition primitives are sufficient to express different compositions that are required to create a secure smart meter design. The higher level tactics have been exercised and utilised multiple times in the case study. Applying these higher level tactics reduce the number of steps to achieve the desired design. The restrictions of each primitives utilized in the higher level tactics are inherited and thus needs to be satisfied before applying the tactic.

Chapter 8

Verified Design to Implementation

In this chapter, I discuss one possible way to turn a verified design in SAM into executable code. I am not claiming that this is the only approach.

Figure 8.1 shows the overall approach. First, the design in SAM is transformed to CapArch, which is an extension of xADL (Dashofy et al., 2005) and will be described in Section 8.1. I also transform the verification procedure, written in Datalog, into a taint analysis specification as described in Section 8.1.1. Taint analysis will be used to analyze the CapArch model of the design. Finally, the model in CapArch will be translated into CAmkES (Kuz et al., 2007), as outlined in Section 8.2. CAmkES is a component-based framework that provides support for implementation of componentised systems on seL4 (Klein et al., 2009).

8.1 SAM to CapArch

CapArch is an Architecture Description Language (ADL) that extends xADL with the concept of capabilities. An ADL is a vocabulary, graphical or textual, used to model a system (Bass et al., 2012). The basic structural elements of an ADL are components and connectors. Components represent the main elements in an architecture that perform

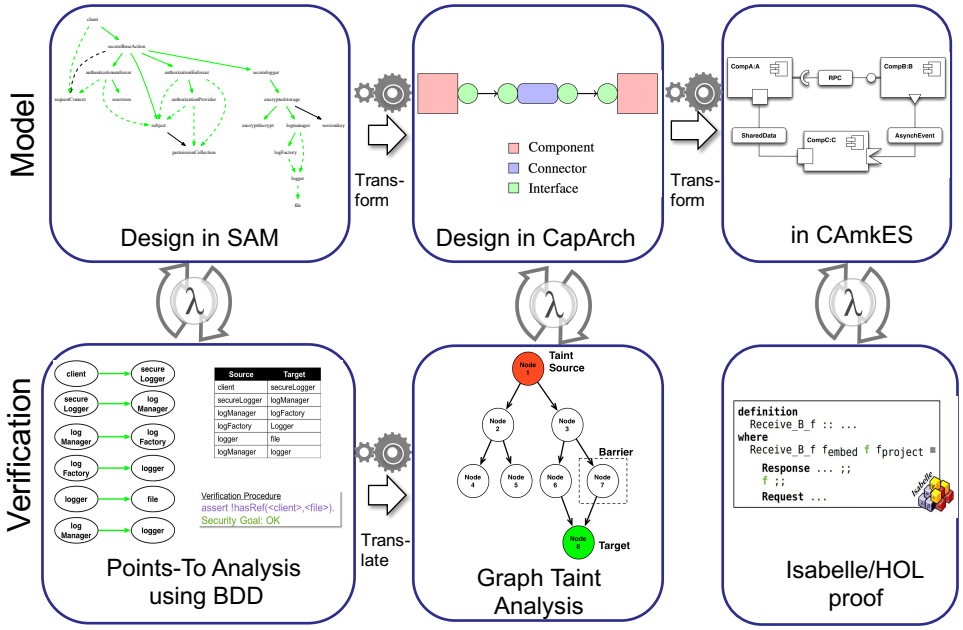


Figure 8.1: Verified design to executable code process. The design in SAM is transformed to CapArch. This is then transformed to CAmkES, which provides a framework to execute code on an underlying platform. The Datalog rule is translated to CapArch taint analysis specification.

computation and stores data, while connectors represent interactions between components. In a graphical representation of an architecture design, a component is represented as a box and a connector as a line connecting components.

In CapArch, there are two basic elements, components and connectors. Each component has a type and may have multiple instances of the component type. Same goes for each connector. A component type has several attributes: *name*, list of *interfaces* and *description* (optional). This is summarized in Table 8.1. Each interface has four attributes: *interfaceID*, *interfaceName*, *interfaceType* and *direction*. *interfaceID* and *interfaceName* are strings. *interfaceType* can either be function, message and data. The function type is realised as Remote Procedure Call (RPC). The message type is realised as asynchronous calls. The data type means that the interface can only transfer data. Direction can be *in*, *out* and *inout*. This is summarized in Table 8.2.

A connection consists of four parameters: *source component*, *interfaceID of source*

Table 8.1: Attributes of a component in CapArch

Attributes	Description
Name	a unique identifier of a component
Description (optional)	description
Interfaces	points of entry/contact

Table 8.2: Attributes of an interface in CapArch

Attributes	Description
id	a unique identifier of an interface
name	a name
type	different type of interface
direction	message flow direction

component, *target component*, *interfaceID of target component*. CapArch requires that the direction of the interfaces of the source component and target component match. This is summarized in Table 8.3.

Table 8.3: Attributes of a connection in CapArch

Attributes	Description
source component	the name of the source component
interfaceID of source component	a unique id of an interface of the source component
target component	the name of the target component
interfaceID of target component	a unique id of an interface of the target component

CapArch has existing tools to translate from CapArch to code that runs on CAmkES, which will be discussed in Section 8.2.

In order to transform a SAM model into a CapArch model specification, several pieces of information are needed. The transformation requires a list of components, including a set of interfaces for each component, and a list of connections. This information can be extracted from a SAM model. I have implemented a tool to automate such extraction and translation of SAM to CapArch.

The component name can be easily extracted from a SAM model. However, extracting the interfaces of each component is more challenging as this information is not directly available in a SAM model. The data in SAM's `hasRef` relationship, which

specifies other components that each component has access to, is used to provide the relevant information. The `hasRef` relationship has two parameters: source and target components. This relationship also implies a uni-directional relationship: from source to target. A list of interfaces can be built with this information. One entry in the `hasRef` relationship allows us to create two interfaces, one for the source component and another for the target component. Figure 8.2 shows this mapping and identifies which pieces of information are still missing.

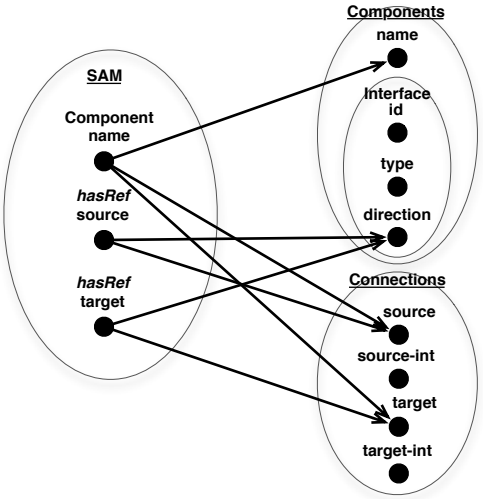


Figure 8.2: SAM to CapArch fields mapping

Since component, interface and connection are the three main elements in a CapArch model, a class for each of these elements is created. The component class captures the name — name of the component instance, type — name of the component type, and a list of interfaces of a component. The interface class will capture *interfaceID*, *interfaceName*, *interfaceType* and *direction*. By default, *interfaceID* is set with the same value as the *interfaceName* and *interfaceType* is set as function. A connection class captures the name of a source component, the *interfaceID* of a source component, the name of a target component and the *interfaceID* of a target component.

For the interface of a source component, its attributes are set to:

- `interfaceID = "I" + sourceName + "out"`
- `interfaceName = "I" + sourceName + "out"`
- `interfaceType = function`
- `direction = out`

For the interface of a target component, its attributes are set to:

- `interfaceID = "I" + targetName + "in"`
- `interfaceName = "I" + targetName + "in"`
- `interfaceType = function`
- `direction = in`

After each interface is created, a connection between these two components and interfaces can be created. Recall that a connection has four parameters: source component, id of source interface, target component and id of target interface. This connection is created with these parameters:

- `source_componentName = sourceName`
- `source_interfaceID = "I" + sourceName + "out"`
- `target_componentName = targetName`
- `target_interfaceID = "I" + targetName + "in"`

In CapArch, a component is assumed to have a connection to itself. This simplifies an architecture designed in CapArch as there is no connection to self required. A connection in CapArch represents a capability in SAM. As there is no connection from a component to itself, I filter out the capabilities from a component to itself when parsing the `hasRef` relationship, i.e. ignore if source component is equal to target component.

As an example, I will transform the secure logger design fragment, which is presented in Section 4.5, into a CapArch model. Tables 8.4 and 8.5 show the `isA` and `hasRef` relationship respectively. I will focus specifically on the *logManager* `hasRef` to *logManager* relationship, as highlighted in Table 8.5. In this relationship, *logMan-*

Table 8.4: SecureLogger—*isA* Relationship

Object *(component instance)*	**Class** *(component type)*
file	File
logFactory	LogFactory
logManager	LogManager
logger	Logger
secureLogger	SecureLogger
client	Client

Table 8.5: SecureLogger—*hasRef* Relationship

Source	**Target**
file	file
logManager	logManager
logManager	logFactory
logManager	logger
...	...
...	...
client	secureLogger

ager is the source and *logger* is the target. First, these two components are created in CapArch. Then, an interface for *logManager* is created and attached to *logManager*. The attributes for this interface are set as follows:

- `interfaceID = "IlogManagerout"`
- `interfaceName = "IlogManagerout"`
- `interfaceType = function`
- `direction = out`

An interface for *logger* is created with its attributes set as follows:

- `interfaceID = "Iloggerin"`
- `interfaceName = "Iloggerin"`
- `interfaceType = function`
- `direction = in`

After both components and their respective interfaces are created, a connection between these two interfaces is created with the following the parameters:

- `source_componentName = logManager`
- `source_interfaceID = "IlogManagerout"`
- `target_componentName = logger`
- `target_interfaceID = "Iloggerin"`

Figure 8.3 summarizes this, showing two components (*logManager* and *logger*) and a connection between them. The top part of the figure shows the values for each of the parameters of the components while the bottom part shows the connection from *logManager* to *logger*.

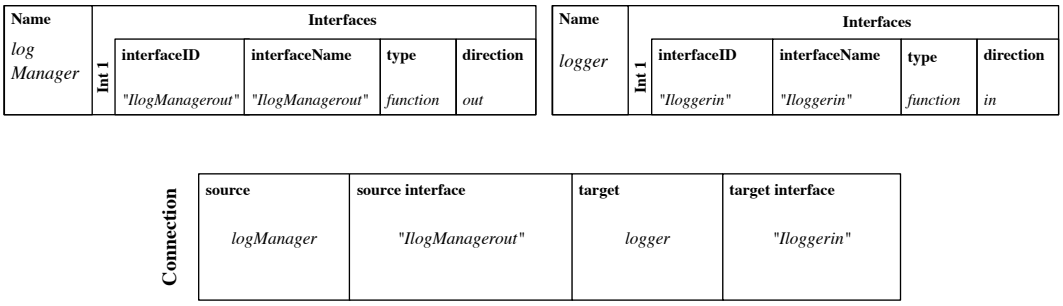


Figure 8.3: An overview of SAM to CapArch Translation

8.1.1 Verification Procedure to Taint Analysis

CapArch uses taint analysis to check whether an architecture has desired security properties. Taint analysis (Schwartz et al., 2010) is a static code analysis technique to detect security vulnerabilities by checking whether information can flow from one node to a target node. A taint is used to mark each node that it has traversed through. If the taint reaches the target node, a security vulnerability is identified. The analysis allows nodes to act as barriers and block taint. If a taint flows through a barrier, that taint is blocked.

In order to perform a taint analysis, the flow of information, barriers and the taints needs to be specified and written into a specification. The taint analysis specification in CapArch has three main elements: *taint*, *query* and *barrier*. *Taint* identifies the source component of a particular flow, tracked with a taint. *Query* is used to check whether a particular *taint* reaches a target component, usually a secret to be protected. *Barrier* specifies the components that act as barriers and block taint. A barrier is usually a trusted component. Each *taint* is specified with a name and a set of components as its sources.

A *query* is specified as the name of a taint and the target component. A *barrier* specifies the name of a component that acts as a barrier and the name of the taint(s) it blocks. This is summarized in Table 8.6.

The verification procedure of a design is translated into a CapArch taint analysis specification. This starts with translating the Datalog rules into *taints* before constructing the *query* and *barrier*. Recall that the template requires source components, target components and (optional) exclusions. Figure 8.4 shows the mappings of elements from a verification procedure to CapArch taint analysis. The name and source of a verification procedure are used as the name and source of a *taint*. For a *query*, the name and the target component of a verification procedure are used as the name and target component in a *query* respectively. Finally, the set of trusted components in a verification procedure is used as the blocking component(s) of a *barrier*.

As an example, I take the verification procedure of the secure logger design fragment, as shown in Listing 8.1, and translate that to CapArch taint analysis. Four different elements are extracted from Listing 8.1: name, source, target and trusted. The components in the secure logger design fragment specified in Section 4.5 are *client*, *secureLogger*, *logManager*, *logFactory*, *logger* and *file*. The secret to be protected (i.e. target) is *file*. Both *logManager* and *logger* are trusted to have access to *file*. The remaining components are part of source in the Verification Procedure. The following shows the value for

Table 8.6: The elements and attributes in a CapArch taint analysis specification

Elements	**Attributes**	**Description**
Taint	name	a unique identifier of the taint
	source component	the source of the taint
Query	taint name	the taint to be checked
	target component	the component to be checked whether the taint reached it
Barrier	name	a unique identifier of the barrier
	taint name	the name of the taint to be blocked by the barrier

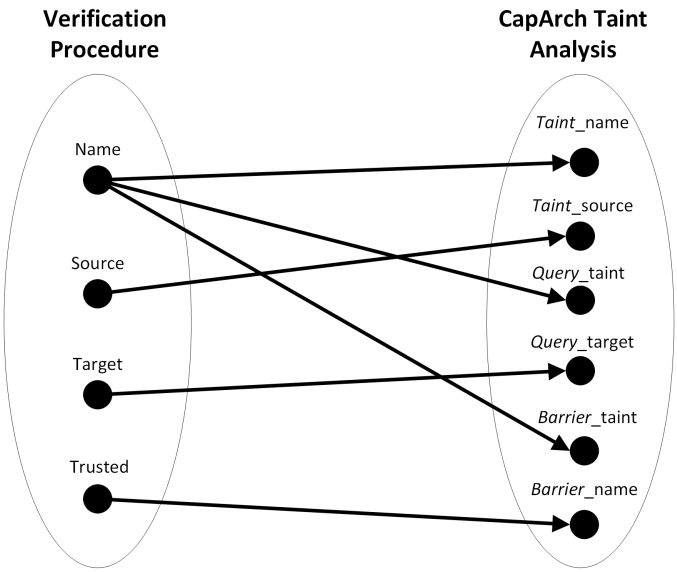


Figure 8.4: Verification Procedure to CapArch Taint Analysis mapping

each elements of the verification procedure.

- Name: fileBreached
- Source: client, secureLogger, logFactory
- Target: file
- Trusted: logManager, logger

Listing 8.1: Secure Logger

```
1 fileBreached(?Src,?T) :-
2     !MATCH(?Src, ?T),
3     !MATCH(?Src,<logManager>),
4     !MATCH(?Src,<logger>),
5     hasRef(?Src,?T).
6 assert !fileBreached(?Src,<file>).
```

Using the mapping shown in Figure 8.4, Listing 8.1 is translated into a CapArch taint analysis specification. The name of the verification procedure is used as the name in *taint*, taint name in *query* and taint name in *barrier*. The source in the verification

procedure is translated as the source components of the taint. The target component in the verification procedure is translated as the target component in *query*. Finally, each of the trusted components in the verification procedure is translated as a separate barrier in *barrier*. The resulting CapArch taint analysis specification is shown below. Note that in the taint analysis specification, the target component in *query* is written as `instance: targetComponent`.

```
taints:
  - name: fileBreached
    links_to: [ClientInst,SecureLoggerInst,LogFactoryInst]
queries:
  - taint: fileBreached
    instance: FileInst
barriers:
    - name: LogManagerInst
      blocks: [fileBreached]
    - name: LoggerInst
      blocks: [fileBreached]
```

8.2 CapArch to CAmkES

In order to get to executable code, the CapArch model is translated to CAmkES. CAmkES is a component-based platform that abstracts the low-level mechanisms of a microkernel (more specifically seL4) and allows modelling systems as a collection of interacting components, generating the code (also known as “glue” code) for communication between components. I use a pre-existing tool to perform the translation.

A CAmkES specification consists of 3 parts: component type specification, com-

ponent instantiation and connection specification. The component type specification defines the details of each component, including which interfaces it provides and uses. The component instantiation part defines instances of the component types. The connection specification outlines the specifics of each connection between components in the system. Note that a CAmkES specification sets up the structure for the system. System initialisation and communication glue code is generated based on the specification but the actual code for each component will then need to be programmed manually.

For each component in a CapArch model, a component type is created in CAmkES. Each component type has a list of interfaces that it provides and uses. This information can be extracted from the list of connections that exist in a CapArch model. Recall that there are four parameters for a connection in a CapArch model: *source component*, *interfaceID of source component*, *target component*, and *interfaceID of target component*. Each connection provides two pieces of information: 1) the interface that *target component* provides; and 2) the interface that *source component* expects that the *target component* provides. Extracting this information from each of the connections in a CapArch model provides sufficient information for the component type specification. One additional piece of information that is required is which component type initiates the model. This is usually the client or the initiator in a design fragment as described in Section 4.4.

After the component type specification is completed, an instance for each component type is created. Then, the connection specification is built. A connection in CAmkES has six parameters: *name*, *type*, *source component*, *source interface*, *target component* and *target interface*. The name of a connection can be automatically generated (currently it is the concatenation of the names of source component and target component). The type of a connection can be extracted from the type of interfaces involved. If the type

is function, then the seL4RPC[1] is used as the connection type. If the type is message, seL4Async[2] is used as the connection type. The type in each end of a connection, i.e. the type that is used by *source interface* and *target interface*, needs to match. Finally, the last four parameters can be extracted directly from the connection list in a CapArch model. This is summarized in Table 8.7.

As an example, I translate the secure logger model in CapArch to CAmkES, focusing on the *logManager* and *logger* components. Figure 8.5 shows the mapping from the model in CapArch to CAmkES. The left side of the figure shows the specification for both components and the connection between them in CapArch.

The first step to translate CapArch to CAmkES is to create the component type specification. One component type is created for *logManager* and another for *logger*. Then, two pieces of information are extracted using the specified connection in CapArch: 1) *logger* provides an interface called "Iloggerin"; and 2) *logManager* uses "Iloggerin". This is shown in the top right part of Figure 8.5. The next step is to create component instances of these types: *logManager* and *logger*. Finally, the connection specification for CAmkES is built. A connection in CAmkES requires six parameters, which are set as follows:

Table 8.7: Attributes of a connection in CAmkES

Attributes	Description
name	a unique identifier of a connection
type	the type of connection. seL4RPC/seL4Async
source component	the name of the source component
source interface	the interface of the source component
target component	the name of the target component
target interface	the interface of the target component

[1] seL4RPC — a connection type specific to seL4 for function calls

[2] seL4Async — a connection type specific to seL4 for asynchronous communication. Please refer to seL4 reference manual for more information, available at https://sel4.systems/Docs/seL4-manual.pdf

- `Name: logManager-logger`
- `Type: seL4RPC`
- `Source: logManager`
- `Source Interface: IlogManagerout`
- `Target: logger`
- `Target Interface: Iloggerin`

The name is a concatenation of *logManager* (source component) and *logger* (target component). The connection type is seL4RPC as both the interfaces have *function* as their type in CapArch. The next four parameters are extracted directly from the CapArch connection.

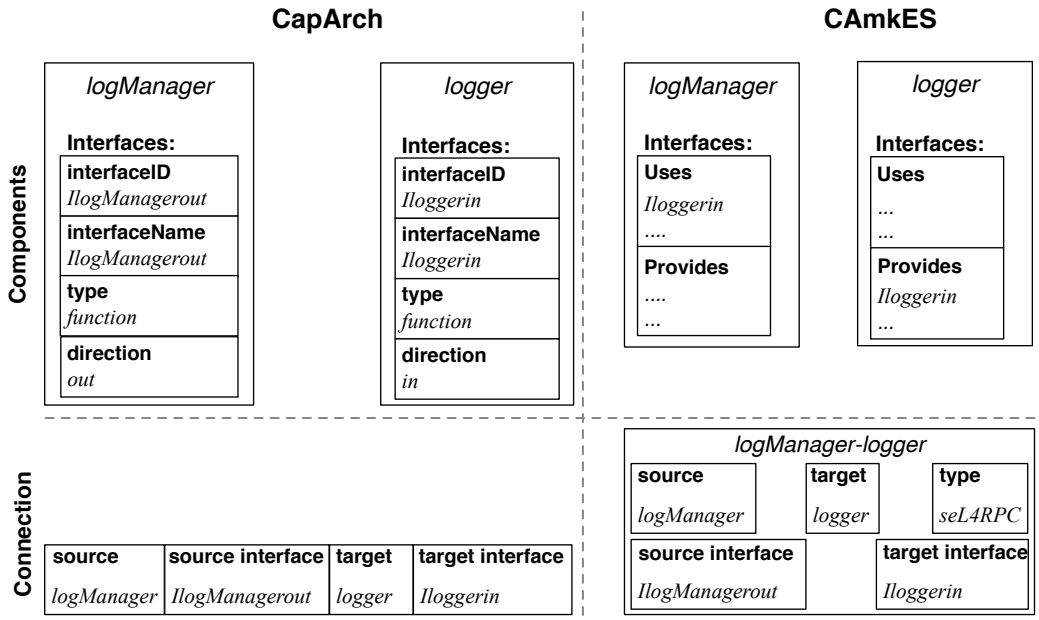


Figure 8.5: CapArch to CAmkES Translation

The result of this translation is a CAmkES specification, which provides the structure of the application. In order to produce executable code, the implementation code that is written in C is required for each of the components. In addition to the implementation code, the required CAmkES libraries and seL4 are needed before being able to build the

application and run it on top of the microkernel.

8.3 Discussion

This chapter provides evidence that it is feasible to implement our designs constructed using the approach proposed in this thesis. I transform the SAM model of the secure logger design fragment, which is presented in Section 4.5, to a CapArch model. The verification procedure of the secure logger design fragment is then translated into a CapArch Taint Analysis specification. Furthermore, the CapArch model is transformed into a CAmkES specification, which provides the structure of the system. Finally, I write the implementation code for each of the components and execute secure logger on top of seL4.

The model transformation from SAM to CapArch has not been formally verified for correctness. I rely on the taint analysis performed on the CapArch specification to provide assurance that the transformation is as intended. Note, however, that the translation from verification procedure to CapArch taint analysis specification has also not been formally verified for correctness. At this stage, I manually inspect the translated taint analysis specification to determine whether it reflects the property that is intended. Formally verifying both the model transformation and verification procedure translation remains as future work.

The model transformation from CapArch to CAmkES has not been formally verified for correctness. I manually inspect the CAmkES specification to ensure that the structure of the system reflects that of the CapArch specification. Formally verifying this transformation remains as future work. Furthermore, there is currently no process to translate a CapArch taint analysis specification into Isabelle/Higher Order Logic (HOL). Isabelle (Nipkow et al., 2002) is an interactive Higher-Order Logic theorem prover. Generating

a proof for correctness of a CAmkES specification and the generated "glue" code is still work in progress (Fernandez et al., 2015, 2013). This proof can then be used together with the proof of the underlying platform (Klein et al., 2014; Murray et al., 2013) to provide a whole system proof. Furthermore, a formal correspondence between the semantics of the verified design and the underlying platform needs to be established in order to formally prove the soundness of the transformations. This remains as future work.

Chapter 9

Conclusion and future work

In this thesis, I have presented a pattern-based composition approach to build high assurance secure application design using capability-based design fragments which specialise patterns for a specific platform.

I defined the following research questions in this thesis:

RQ-1: How can I specialize security patterns as reusable design fragments targeting particular platforms?

RQ-2: How can I verify that security properties hold of these design fragments?

RQ-3: How can many design fragments be applied to the design and verification of secure software systems?

This thesis has addressed all the research questions defined above and has made the following contributions, each of which addressed one of the research questions:

A New Security Patterns Catalog

In Chapter 3, I collected 279 security patterns from the existing literature into a security pattern catalog. This provides a library of solutions to recurring security problems. I

have proposed a new security template that unifies five existing commonly used pattern templates. With this new template, I am able to collect relevant information for security patterns.

Capability-specific Design Fragments

Chapter 4 proposes the concept of design fragments, which realize design patterns for a particular platform. I have provided a guideline on deriving design fragments from design patterns. This was illustrated by examples of capability-specific design fragments derived from security patterns. This contribution addresses my first research question (RQ-1).

Composition of Design Fragments

A pattern-based composition approach to incrementally build and verify application designs was presented in Chapter 5. This approach uses verified design fragments and the proposed composition tactics to reuse the design fragments to build an application design and reuse the verification procedures to verify the resulting application design. This approach has been shown to be applicable in two case studies from different domains, Chapter **??** and **??**. The composition tactics have also been proven to be property preserving. This contribution answers our third research question (RQ-3). Finally, a general security property, the *Protected By* property, is defined in this chapter.

Verification Procedure

In Chapter 6, the concept of verification procedure is introduced. This captures information about a system which allows analysis to check security properties. I defined a template to capture security properties and showed that this template is applicable in two different case studies. This contribution answers my second research question (RQ-2) and partially RQ-3.

9.1 Discussion

The case studies have shown that our approach is applicable in different domains. DevOps Continuous Deployment pipelines and embedded systems (i.e. Smart Meter) are substantially different technology domains and application domains. The capability-based design fragments and their verification procedure are reused during several compositions in the case studies. This is intended to not only reduce the design effort but also the verification effort. Furthermore, the case studies have also indicated that our composition tactics, which are defined in Chapter 5, are sufficient to express the intended compositions. I have not, however, formally proven the completeness of these tactics.

One of the common pitfalls for model-checking verification is lack of scalability, due to state explosion. However, I perform design-level verification, which is on smaller models compared to code-level verification. Berndl et al. (2003) have demonstrated that Points-To analysis using BDD can scale to verify code of large systems. Furthermore, I perform incremental checking of properties which reduces the state search space.

A potential weakness of design-level verification is that the security property might break at the code level or physical level. This can be due to misunderstanding the design, coding errors, or to implementations exposing information that had been abstracted in the design. In a componentized system, such as the one developed in the smart meter case study, a lot of 'glue code' is used to implement communication between components. Writing such code manually can introduce many errors. One way of mitigating this is to utilize a component-based development framework to generate the 'glue code'. Besides avoiding typical coding mistakes, using such a framework can provide high-level assurance, assuming that the generated code comes with assurance that the refinement to code does not break the design-level security property. The component code must also be correct and appropriate assurance of its correctness is required.

Attacks at the physical level can also violate assumptions made by design abstrac-

tions. For example, covert channel attacks can break confidentiality properties. Alternative mitigations and arguments are required to provide assurance that a system can resist such attacks, but I have not directly addressed this issue in this dissertation.

Currently, all of our design fragments are verified using the same verification approach. I have not experimented with composing design fragments that are verified using different techniques.

A design fragment is defined as a partial realization of a design pattern in the context of a particular platform. I have shown realizations of security patterns for one particular platform, i.e. capability-based platform. I believe that this concept can be used to realize patterns for other platforms as well but have not yet experimented with that.

Security patterns are the source for the definition of capability-specific design fragments. However, as patterns are informal and ambiguous, it can be difficult to know exactly what properties are being claimed and what should be verified. Moreover, it is difficult to compare a formal representation of the pattern as a design fragment, to an informal textual one. A design fragment for a security pattern is one version of the pattern specialized for a specific platform. I do not claim that a design fragment of a pattern for a platform is the only possible representation of the pattern for that platform.

The transformation approach to take verified design to runnable code, described in Chapter 8, is one possible way to produce runnable code from a verified design. At this stage, I rely on expert opinion that the model transformation is correct, and perform a taint analysis to prove the security properties of the transformed model. I have not formally verified the model transformations for correctness.

9.2 Future work

Composition Tactic Completeness

The current collection of composition primitives, presented in Chapter 5, has been shown to be sufficient to express different kinds of composition that are required in both case studies. Since these case studies are from different domains, they indicate that the tactics are complete enough. An interesting piece of future work will be to further explore this issue. One way to widen the coverage of the tactics is to relax their restrictions such that the tactics can express more design steps and still preserve the *Protected By* property.

Model Transformation Correctness Proof

The transformation approach can turn verified design into executable code. At this stage, it relies on expert opinion that the model transformation is correct, and a taint analysis to prove the security properties of the transformed model. Interesting future work will be to formally prove that the model transformation is correct. One possible start is to provide a mapping between two models and proof that the mapping is correct. Then, verify that the code that implements the transformation is correct. This requires code-level verification and significant effort.

Formal correspondence of semantics

In order to formally prove the soundness of the model transformations presented in Chapter 8, a formal correspondence between the semantics of the verified design and the underlying platform is required. The aim is to preserve the verified properties of the design during the transformation to executable code. This is an interesting piece of future work to be explored.

Specialize Design Fragment for other Platforms

The collection of design fragments presented in this thesis currently targets platforms

that adhere to the capability-based security model (Dennis and Van Horn, 1966). Specializing design fragments for other platforms can be an interesting direction to pursue.

Bibliography

Alexander, C., Ishikawa, S. and Silverstein, M. (1977), *A pattern language: towns, buildings, construction*, Vol. 2, Oxford University Press.

Alvi, A. K. and Zulkernine, M. (2011), A natural classification scheme for software security patterns, in *Proceedings of the 2011 IEEE 9th International Conference on Dependable, Autonomic and Secure Computing*, IEEE Computer Society, Washington, DC, USA, pp. 113–120.

Andersen, L. O. (1994), Program analysis and specialization for the C programming language, PhD thesis, University of Cophenhagen.

Anderson, M., Pose, R. D. and Wallace, C. S. (1986), 'A password-capability system', *Computer* **29**, 1–8.

Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. (2004), 'Basic concepts and taxonomy of dependable and secure computing', *IEEE Trans. Dependable Secur. Comput.* **1**(1), 11–33.

Barker, E. B., Barker, W. C., Burr, W. E., Polk, W. T. and Smid, M. E. (2007), Recommendation for Key Management, Technical Report SP 800-57, National Institute of Standards and Technology, Gaithersburg, MD, United States.

Bass, L., Clements, P. and Kazman, R. (2012), *Software Architecture in Practice*, 3rd edn, Addison-Wesley Professional, Boston, MA, USA.

Bass, L., Holz, R., Rimba, P., Tran, A. B. and Zhu, L. (2015), Securing a deployment pipeline, in *Proceedings of the 3rd International Workshop on Release Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 4–7.

Bass, L., Weber, I. and Zhu, L. (2014), *DevOps: A Software Architect's Perspective*, Addison-Wesley Professional, Boston, MA, USA.

Bayley, I. and Zhu, H. (2008), On the composition of design patterns, in *Proceedings of the 8th International Conference on Quality Software*, IEEE Computer Society, Washington, DC, USA, pp. 27–36.

Berndl, M., Lhoták, O., Qian, F., Hendren, L. and Umanee, N. (2003), Points-to analysis using BDDs, in *Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, New York, NY, USA, pp. 103–114.

Bishop, P. and Bloomfield, R. (1998), A methodology for safety case development, in F. Redmill and T. Anderson, eds, *Industrial Perspectives of Safety-critical Systems*, Springer London, pp. 194–203.

Blakley, B. and Heath, C. (2004), Security Design Patterns, Technical report, The Oopen Group Security Forum.

Bloomfield, R. and Bishop, P. (2010), Safety and assurance cases: Past, present and possible future — an Adelard perspective, in C. Dale and T. Anderson, eds, *Making Systems Safer*, Springer London, pp. 51–67.

Bogdanov, A., Khovratovich, D. and Rechberger, C. (2011), Biclique cryptanalysis of the full AES, in *Proceedings of the 17th international conference on The Theory and*

Application of Cryptology and Information Security, Springer-Verlag, Berlin, Heidelberg, pp. 344–371.

Braga, A. M., Rubira, C. M. F. and Dahab, R. (1998), Tropyc: A pattern language for cryptographic software, in *Proceedings of the 5th Conference on Pattern Languages of Programs*, pp. 25:1–25:27.

Cai, L., Machiraju, S. and Chen, H. (2010), CapAuth: A Capability-based Handover Scheme, in *Proceedings of the IEEE INFOCOM 2010*, IEEE Press, pp. 1 –5.

Castro, M. D., Pose, R. D. and Kopp, C. (2008), 'Password-capabilities and the walnut kernel', *Computer* **51**(5), 595–607.

Ceri, S., Gottlob, G. and Tanca, L. (1989), 'What you always wanted to know about datalog (and never dared to ask)', *IEEE Trans. on Knowl. and Data Eng.* **1**(1), 146–166.

Dashofy, E. M., Hoek, A. v. d. and Taylor, R. N. (2005), 'A comprehensive approach for the development of modular software architecture description languages', *ACM Trans. Softw. Eng. Methodol.* **14**(2), 199–245.

Delessy-Gassant, N., Fernandez, E. B., Rajput, S. and Larrondo-Petrie, M. M. (2004), Patterns for application firewalls, in *Proceedings of the 11th Conference on Pattern Languages of Programs*, pp. 23:1–23:19.

Delessy, N. and Fernandez, E. B. (2005), Patterns for the eXtensible Access Control Markup Language, in *Proceedings of the 12th Conference on Pattern Languages of Programs*, pp. 7–10.

Delessy, N., Fernandez, E. B., Larrondo-Petrie, M. M. and Wu, J. (2007), Patterns for access control in distributed systems, in *Proceedings of the 14th Conference on Pattern Languages of Programs*, ACM, New York, NY, USA, pp. 3:1–3:11.

Dennis, J. B. and Van Horn, E. C. (1966), 'Programming semantics for multiprogrammed computations', *Commun. ACM* **9**(3), 143–155.

Dong, J., Peng, T. and Zhao, Y. (2010), 'Automated verification of security pattern compositions', *Inf. Softw. Technol.* **52**(3), 274–295.

Dougherty, C. R., Sayre, K., Seacord, R., Svoboda, D. and Togashi, K. (2009), Secure design patterns, Technical Report CMU/SEI-2009-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Dumitraş, T. and Narasimhan, P. (2009), Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system, in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 18:1–18:20.

Dyson, P. and Longshaw, A. (2003), Patterns for Managing Internet-Technology Systems, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 459–492.

Elsinga, B. and Hofman, A. (2002), Control the actor-based access rights, in *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pp. 233–244.

Elsinga, B. and Hofman, A. (2003), Security taxonomy pattern language, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 18:1–18:12.

F. Lee Brown, J., DiVietri, J., Villegas, G. D. d. and Fernandez, E. B. (1999), The authenticator pattern, in *Proceedings of the 6th Conference on Pattern Language of Programs*, pp. 15–18.

Fernandez-Buglioni, E. (2013), *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, 1st edn, Wiley Publishing, Hoboken, NJ, USA.

Fernandez, E. B. (2002), Patterns for operating system access control, in *Proceedings of the 9th Conference on Pattern Languages of Programs*, pp. 12:1–12:13.

Fernandez, E. B. (2004), Two patterns for web services security, in *Proceedings of the 2nd International Symposium on Web Services and Applications*, pp. 801–807.

Fernandez, E. B., Larrondo-Petrie, M. M., Seliya, N., Delessy, N. and Herzberg, a. A. (2003), A pattern language for firewalls, in *Proceedings of the 10th Conference on Pattern Languages of Programs*, pp. 6:1–6:13.

Fernandez, E. B. and Pan, R. (2001), A pattern language for security models, in *Proceedings of the 8th Conference on Pattern Languages of Programs*, pp. 14:1–14:13.

Fernandez, E. B. and Sinibaldi, J. C. (2003), More patterns for operating system access control, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 381–398.

Fernandez, E. B., Sorgente, T. and Larrondo-Petrie, M. M. (2006), Even more patterns for secure operating systems, in *Proceedings of the 13th Conference on Pattern Languages of Programs*, ACM, New York, NY, USA, pp. 10:1–10:9.

Fernandez, E. B. and Warrier, R. (2003), Remote authenticator/authorizer, in *Proceedings of the 10th Conference on Pattern Languages of Programs*, pp. 8:1–8:8.

Fernandez, M., Andronick, J., Klein, G. and Kuz, I. (2015), Automated verification of RPC stub code, in N. Bjørner and F. de Boer, eds, *FM 2015: Formal Methods*, Vol. 9109 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 273–290.

Fernandez, M., Kuz, I., Klein, G. and Andronick, J. (2013), Towards a verified component platform, in *Proceedings of the 7th Workshop on Programming Languages and Operating Systems*, ACM, New York, NY, USA, pp. 2:1–2:7.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Graydon, P. J., Knight, J. C. and Strunk, E. A. (2007), Assurance based development of critical systems, in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE Computer Society, Washington, DC, USA, pp. 347–357.

Grove, D., Murray, T., Owen, C., North, C., Jones, J., Beaumont, M. and Hopkins, B. (2007), An overview of the Annex system, in *Proceedings of the Annual Computer Security Applications Conference*, Miami Beach, Florida, pp. 341–352.

Hafiz, M. (2006), A collection of privacy design patterns, in *Proceedings of the 13th Conference on Pattern Languages of Programs*, ACM, New York, NY, USA, pp. 7:1–7:13.

Hafiz, M., Adamczyk, P. and Johnson, R. E. (2007), 'Organizing security patterns', *IEEE Softw.* **24**(4), 52–60.

Hafiz, M., Adamczyk, P. and Johnson, R. E. (2012), Growing a pattern language (for security), in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM, New York, NY, USA, pp. 139–158.

Hardy, N. (1985), 'KeyKOS architecture', *SIGOPS Oper. Syst. Rev.* **19**(4), 8–25.

Hardy, N. (1988), 'The confused deputy: (or why capabilities might have been invented)', *SIGOPS Oper. Syst. Rev.* **22**(4), 36–38.

Hasheminejad, S. M. and Jalili, S. (2009), Selecting Proper Security Patterns Using Text Classification, in *Proceedings of the International Conference on Computational Intelligence and Software Engineering*, IEEE, pp. 1–5.

Hays, V., Loutrel, M. and Fernandez, E. B. (2000), The Object Filter and Access Control Framework, in *Proceedings of the 7th Conference on Pattern Languages of Programs*, pp. 12–17.

Heyman, T., Scandariato, R. and Joosen, W. (2012), Reusable formal models for secure software architectures, in *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, IEEE Computer Society, Helsinki, Finland, pp. 41–50.

Heyman, T., Yskout, K., Scandariato, R. and Joosen, W. (2007), An analysis of the security patterns landscape, in *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, IEEE Computer Society, Washington, DC, USA, pp. 3–9.

Holzmann, G. (1997), 'The model checker SPIN', *IEEE Trans. Softw. Eng.* **23**(5), 279–295.

Jackson, D. (2012), *Software Abstractions: Logic, Language, and Analysis*, The MIT Press.

Joint Task Force Transformation Initiative (2010), Recommended Security Controls for Federal Information Systems and Organizations, Technical Report SP 800-53, National Institue of Standards and Technology, Gaithersburg, MD, United States.

Kelly, T. P. (1999), Arguing Safety — A Systematic Approach to Safety Case Management, PhD thesis, University of York.

Kelly, T. and Weaver, R. (2004), The goal structuring notation–a safety argument notation, in *Proceedings of the Dependable Systems and Networks 2004 workshop on assurance cases*.

Kienzle, D. M. and Elder, M. C. (2002), Final technical report: Security patterns for web application development, Technical report, DARPA.

Kitchenham, B. (2004), Procedures for performing systematic reviews, Technical Report 2004, Keele, UK, Keele University.

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. and Heiser, G. (2014), 'Comprehensive formal verification of an os microkernel', *ACM Trans. Comput. Syst.* **32**(1), 2:1–2:70.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S. (2009), sel4: Formal verification of an os kernel, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, ACM, New York, NY, USA, pp. 207–220.

Kodituwakku, S. R., Bertok, P. and Zhao, L. (2001), APLRAC: A pattern language for designing and implementing role-based access control, in *Proceedings of the 6th European Conference on Pattern Languages of Programs*, pp. 331–346.

Konrad, S., Cheng, B. H. C., Campbell, L. A. and Wassermann, R. (2003), Using security patterns to model and analyze security requirements, in *Proceedings of the Requirements Engineering for High Assurance Systems (RHAS)*, IEEE.

Kubo, A., Washizaki, H. and Fukazawa, Y. (2007), Extracting relations among security patterns, in *Proceedings of the 1st International Workshop on Software Patterns and Quality*.

Kuz, I., Liu, Y., Gorton, I. and Heiser, G. (2007), 'Camkes: A component model for secure microkernel-based embedded systems', *J. Syst. Softw.* **80**(5), 687–699.

Kuz, I., Zhu, L., Bass, L., Staples, M. and Xu, X. (2012), An architectural approach for cost effective trustworthy systems, in *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, IEEE Computer Society, Helsinki, Finland, pp. 325–328.

Lampson, B. W. (1973), 'A note on the confinement problem', *Commun. ACM* **16**(10), 613–615.

Laverdiere, M.-A., Mourad, A., Hanna, A. and Debbabi, M. (2006), Security design patterns: Survey and evaluation, in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pp. 1605–1608.

Lehtonen, S. and Pärssinen, J. (2001), A Pattern Language for Cryptographic Key Management, in *Proceedings of the 9th Conference on Pattern Languages of Programs*, pp. 35:1–35:13.

Leonard, T., Hall-May, M. and Surridge, M. (2013), 'Modelling access propagation in dynamic systems', *ACM Trans. Inf. Syst. Secur.* **16**(2), 5:1–5:31.

Levy, H. M. (1984), *Capability-Based Computer Systems*, Butterworth-Heinemann, Newton, MA, USA.

McUmber, W. E. and Cheng, B. H. C. (2001), A general framework for formalizing UML with formal languages, in *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 433–442.

Miller, M. S. (2006), Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, PhD thesis, Johns Hopkins University.

Miller, M. S., Yee, K.-P. and Shapiro, J. (2003), Capability myths demolished, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory.

Morrison, P. and Fernandez, E. B. (2006*a*), The credentials pattern, in *Proceedings of the 2006 Conference on Pattern Languages of Programs*, ACM, New York, NY, USA, pp. 9:1–9:4.

Morrison, P. and Fernandez, E. B. (2006*b*), Securing the broker pattern, in *Proceedings of the 11th European Conference on Pattern Languages of Programs*, pp. 513–530.

Mourad, A., Otrok, H. and Baajour, L. (2010), A novel approach for the development and deployment of security patterns, in *Proceedings of the IEEE Second International Conference on Social Computing*, IEEE Computer Society, Washington, DC, USA, pp. 914–919.

Mouratidis, H., Giorgini, P. and Schumacher, M. (2003), Security patterns for agent systems, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 399–416.

Mullender, S. J., Rossum, G. v., Tanenbaum, A. S., Renesse, R. v. and Staveren, H. v. (1990), 'Amoeba: a distributed operating system for the 1990s', *Computer* **23**, 44–53.

Mullender, S. J. and Tanenbaum, A. S. (1986), 'The Design of a Capability-Based Distributed Operating System', *Computer* **29**(4), 289–299.

Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X. and Klein, G. (2013), sel4: From general purpose to a proof of information flow enforcement, in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, pp. 415–429.

Neumann, P. G. and Watson, R. N. M. (2010), Capability revisited: A holistic approach to bottom-to-top assurance of trustworthy systems, in *Proceedings of the Fourth Annual Layered Assurance Workshop*, Texas, USA, pp. 1–10.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M. and Deardeuff, M. (2015), 'How Amazon Web Services uses formal methods', *Commun. ACM* **58**(4), 66–73.

Nipkow, T., Wenzel, M. and Paulson, L. C. (2002), *Isabelle/HOL: A Proof Assistant for Higher-order Logic*, Springer-Verlag, Berlin, Heidelberg.

Ortiz, R., Moral-García, S., Moral-Rubio, S., Vela, B., Garzás, J. and Fernández-Medina, E. (2010), Applicability of security patterns, in *On the Move to Meaningful Internet Systems (OTM)*, Vol. 6426 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 672–684.

Pärssinen, J. and Turunen, M. (2002), Pattern language for specification of communication protocols, in *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pp. 259–278.

Petersen, K., Feldt, R., Mujtaba, S. and Mattsson, M. (2008), Systematic mapping studies in software engineering, in *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, British Computer Society, Swinton, UK, pp. 68–77.

Porter, M. F. (1997), Readings in information retrieval, in K. Sparck Jones and P. Willett, eds, *Readings in information retrieval*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter An Algorithm for Suffix Stripping, pp. 313–316.

Riehle, D., Cunningham, W., Bergin, J., Kerth, N. and Metsker, S. (2002), Password

patterns, in *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pp. 279–288.

Rimba, P. (2013), Building high assurance secure applications using security patterns for capability-based platforms, in *Proceedings of the 35th International Conference on Software Engineering*, IEEE Press, Piscataway, NJ, USA, pp. 1401–1404.

Rimba, P. (2015), 'A summary on the 37th international conference on software engineering (icse 2015)', *SIGSOFT Softw. Eng. Notes* **40**(4), 35–35.

Rimba, P., Zhu, L., Bass, L., Kuz, I. and Reeves, S. (2015), Composing patterns to construct secure systems, in *Proceedings of the 11th European Dependable Computing Conference*, IEEE Computer Society, Washington, DC, USA, pp. 213–224.

Rimba, P., Zhu, L., Xu, X. and Sun, D. (2015), Building secure applications using pattern-based design fragments, in *Proceedings of the 34th International Symposium on Reliable Distributed Systems*, IEEE Computer Society, Washington, DC, USA.

Romanosky, S. (2001), Security design patterns, in *Proceedings of the Conference on Pattern Languages of Programs*, pp. 1–19.

Romanosky, S., Acquisti, A., Hong, J., Cranor, L. F. and Friedman, B. (2006), Privacy patterns for online interactions, in *Proceedings of the 13th Conference on Pattern Languages of Programs*, ACM, New York, NY, USA, pp. 12:1–12:9.

Rosado, D. G., Fernandez-Medina, E., Piattini, M. and Gutierrez, C. (2006), A study of security architectural patterns, in *Proceedings of the 1st International Conference on Availability, Reliability and Security*, IEEE Computer Society, Washington, DC, USA, pp. 358–365.

Sadicoff, M., Larrondo-Petrie, M. M. and Fernandez, E. B. (2005), Privacy-aware network client pattern, in *Proceedings of the 12th Conference on Pattern Languages of Programs*, pp. 10:1–10:6.

Saltzer, J. and Schroeder, M. (1975), 'The protection of information in computer systems', *Proceedings of the IEEE* **63**(9), 1278–1308.

Saridakis, T. (2003), Design patterns for fault containment, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 493–520.

Schumacher, M. (2002), Security patterns and security standards, in *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pp. 289–300.

Schumacher, M. (2003), Firewall patterns, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 417–430.

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F. and Sommerlad, P. (2006), *Security Patterns: Integrating Security and Systems Engineering*, Wiley Publishing, Hoboken, NJ, USA.

Schwartz, E. J., Avgerinos, T. and Brumley, D. (2010), All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, pp. 317–331.

Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J. and Klein, G. (2011), seL4 enforces integrity, in *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, Springer, pp. 325–340.

Shapiro, J. S. and Hardy, N. (2002), 'EROS: A principle-driven operating system from the ground up', *IEEE Softw.* **19**(1), 26–33.

Shapiro, J. S., Smith, J. M. and Farber, D. J. (1999), EROS: A fast capability system, in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, ACM, New York, NY, USA, pp. 170–185.

Shiroma, Y., Washizaki, H., Fukazawa, Y., Kubo, A. and Yoshioka, N. (2010), Model-driven security patterns application based on dependences among patterns, in *Proceedings of the 5th International Conference on Availability, Reliability and Security*, IEEE, pp. 555–559.

Sommerlad, P. (2003), Reverse proxy patterns, in *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pp. 431–458.

Sørensen, K. E. (2002), Session patterns, in *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pp. 301–322.

Steel, C., Nagappan, R. and Lai, R. (2005), *Core security patterns: best practices and strategies for J2EE, Web services, and identity management*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Swiderski, F. and Snyder, W. (2004), *Threat Modeling*, Microsoft Press, Redmond, WA, USA.

The Advanced Security Acceleration Project (2010), Security profile for Advanced Metering Infrastructure, Technical report, The NIST Cyber Security Coordination Task Group.

VanHilst, M., Fernandez, E. B. and Braz, F. (2009), 'A multi-dimensional classification for users of security patterns', *Journal of Research and Practice in Information Technology* **41**(2), 87–98.

Viega, J. and McGraw, G. (2011), *Building Secure Software: How to Avoid Security Problems the Right Way*, 1st edn, Addison-Wesley Professional, Boston, MA, USA.

Washizaki, H., Fernandez, E. B., Maruyama, K., Kubo, A. and Yoshioka, N. (2009), Improving the classification of security patterns, in *Proceedings of the 20th International Workshop on Database and Expert Systems Application*, IEEE Computer Society, Washington, DC, USA, pp. 165–170.

Weinstock, C., Lipson, H. F. and Goodenough, J. (2013), 'Arguing security - creating security assurance cases', https://buildsecurityin.us-cert.gov/articles/knowledge/assurance-cases/arguing-security-creating-security-assurance-cases. accessed 15 August 2015.

Weiss, M. (2006), Credential delegation: Towards grid security patterns, in *Proceedings of the 5th Nordic Conference on Pattern Languages of Programs*, pp. 65–70.

Weiss, M. and Mouratidis, H. (2008), Selecting security patterns that fulfill security requirements, in *Proceedings of the 16th IEEE International Requirements Engineering Conference*, IEEE Computer Society, Washington, DC, USA, pp. 169–172.

Whaley, J. (2007), Context-sensitive pointer analysis using binary decision diagrams, PhD thesis, Stanford University.

Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R. and Roe, M. (2014), The cheri capability model: Revisiting risc in an age of risk, in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, IEEE Press, Piscataway, NJ, USA, pp. 457–468.

Yoder, J. and Barcalow, J. (1997), Architectural patterns for enabling application security, in *Proceedings of the 4th Conference on Patterns Language of Programming (PLoP)*, Washington, DC, USA.

Yskout, K., Scandariato, R. and Joosen, W. (2012), Does organizing security patterns focus architectural choices?, in *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, Piscataway, NJ, USA, pp. 617–627.

Zachman, J. et al. (1987), 'A framework for information systems architecture', *IBM Syst. J.* **26**(3), 276–292.

Appendix A

Smart Meter Requirements

Table A.1 lists 53 security requirements that are relevant to the Smart Meter case study, which are described in Chapter **??**. These requirements are obtained from The Advanced Security Acceleration Project (2010).

Table A.1: Requirements that are relevant for the Smart Meter case study

#	Requirement Name	Description
DHS-2.8.2	Management Port Partitioning	AMI components isolate data acquisition from management services
DHS-2.8.3	Security Function Isolation	Separate security from non-security functions.
DHS-2.8.4	Information Remnants	Prevent unauthorized or unintended information transfer via shared system resources.

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.8.5	NIST SP 800-53 SC-5 Denial-of-Service Protection	Protect against or limit effects of DoS attacks
DHS-2.8.6	Resource Priority	Limit the use of resources by priority
DHS-2.8.7	Boundary Protection	Define physical and electronic security boundaries for AMI system along with other applications sharing the same environment.
DHS-2.8.8	Communication Integrity	AMI design shall protect the integrity of electronically communicated info
DHS-2.8.9	Communication Confidentiality	AMI shall protect the confidentiality of electronically communicated info
DHS-2.8.10	Trusted Path	Develop a policy governing the use of crypto for protection of AMI sys info.
DHS-2.8.12	Use of Validated Cryptography	Explicit indication of collaborative computing mechanism must be provided to local users
DHS-2.8.13	Collaborative Computing	Reliably associate security parameters with info exchanged between enterprise Infosys and AMI.

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.8.14	Transmission of Security Parameters	Restrict usage of mobile code techs. Document and monitor them.
DHS-2.8.16	Mobile Code	restrict and monitor usage of VoIP within AMI system.
DHS-2.8.17	Voice-Over Internet Protocol	All external AMI components and communication connections shall be adequately protected from tampering or damage.
DHS-2.8.18	System Connections	Security Roles and Responsibilities must be specified and defined.
DHS-2.8.19	Security Roles	Protect authenticity of device-to-device communication
DHS-2.8.20	Message Authenticity	Automatically mark external data output w/ standard naming conventions to identify special handling.
DHS-2.9.1	Information and Document Management Policy and Procedures	Organization shall develop procedures to facilitate the implementation of the AMI
DHS-2.9.10	Automated Marking	The components of AMI shall automatically mark any external data output using standard name convention

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.14.3	Malicious Code Protection	Employ malicious code protection.
DHS-2.14.4	System Monitoring Tools and Techniques	Components shall detect log and report all security events and system activities to AMI management system.
DHS-2.14.6	Security Functionality Verification	All components shall employ controls which independently and in concert with the AMI management system verify that all security functions within the component are in an on-line/active state.
DHS-2.14.7	Software and Information Integrity	AMI shall monitor and detect unauthorized changes to software firmware data
DHS-2.14.8	Unauthorized Communications Protection	AMI shall implement unauthorized communications protection.
DHS-2.14.9	Information Input Restrictions	Organization should implement security measures to restrict info input to AMI only to authorized user.
DHS-2.14.10	Information Input Accuracy, Completeness, Validity, and Authenticity	Components shall check information for validity and authenticity

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.14.11	Error Handling	Components shall employ controls to identify and handle error conditions w/o providing info that can be exploited by adversaries
DHS-2.15.1	Access Control Policy and Procedures	Organization shall develop access control policy
DHS-2.15.2	Identification and Authentication Policy and Procedures	Organization shall develop identification and authentication policy
DHS-2.15.7	Access Enforcement	Components shall enforce assigned authorizations for controlling access to system.
DHS-2.15.8	Separation of Duties	Components enforce separation of duties thru assigned access authorizations.
DHS-2.15.9	Least Privilege	Principle of Least Privilege should be enforced
DHS-2.15.10	User Identification and Authentication	Components shall uniquely identify and authenticate users
DHS-2.15.11	Permitted Actions without Identification or Authentication	Permitted Actions without identification or authentication

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.15.12	Device Identification and Authentication	AMI shall identify and authenticate specific components before establishing a connection
DHS-2.15.13	Authenticator Feedback	Components shall obfuscate feedback of authentication information to protect information from potential exploits
DHS-2.15.14	Cryptographic Module Authentication	Components shall employ authentication methods for cryptography
DHS-2.15.15	Information Flow Enforcement	AMI shall enforce assigned authorizations for controlling info flow
DHS-2.15.16	Passwords	Components shall support passwords with a level of complexity based on the criticality level of the system
DHS-2.15.17	System Use Notification	Components shall support displaying approved system use notification msg.
DHS-2.15.18	Concurrent Session Control	Components shall limit the number of concurrent sessions for any users.
DHS-2.15.19	Previous Logon Notification	Components shall notify user of last logon timestamp&unsuccessful attempts
DHS-2.15.20	Unsuccessful Login Attempts	Components shall limit number of consecutive invalid access attempts.

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.15.21	Session Lock	Components shall preven further access to the system by initiating session lock after a predetermined period of inactivity
DHS-2.15.22	Remote Session Termination	AMI shall auto terminate remote session after a defined period of inactivity
DHS-2.15.24	Remote Access	Components with remote access shall allow access to be enabled only in accordance with the appropriate policy
DHS-2.15.29	Use of External Information Control Systems	Terms and conditions to access AMI and store/process information should be established
DHS-2.16.2	Auditable Events	Components shall generate audit records
DHS-2.16.3	Content of Audit Records	Components shall generate sufficient and detailed information in audit records
DHS-2.16.4	Audit Storage Capacity	Components shall provide enough storage to store audit records.
DHS-2.16.8	Time Stamps	Provide timestamps for use in audit

Continued on next page

Table A.1 – continued from previous page

#	Requirement Name	Description
DHS-2.16.9	Protection of Audit Information	Protect audit info from unauthorized access modification or deletion
DHS-2.16.12	Auditor Qualification	Auditor qualifications should be specified

Appendix B

Sample Security Patterns Catalog

Authorization Enforcer

- **Intent** Defines the access policy for resources
- **Alias** Authorization (Schumacher et al., 2006), Remote Authorizer (Fernandez and Warrier, 2003)
- **Problem** Components need to verify that each request is properly authorized. A way to control access to resources is needed.
- **Solution** For each active entity, indicate which resources it can access and how.
- **Participants** Client, SecureBaseAction, Subject, PermissionCollection, AuthorizationProvider, AuthorizationEnforcer, AccessStore. Each of these can be viewed as a component
- **Interactions** The Client requests authorization from *SecureBaseAction* and sends *Subject*. *SecureBaseAction* uses the credential information in *Subject* and invokes *AuthorizationEnforcer*'s authorize method. *AuthorizationEnforcer* then requests the permissions of the client from *AuthorizationProvider*. *AuthorizationProvider* retrives permission from *AccessStore*, creates *PermissionCollection*, stores it into *Subject* and returns *Subject* to *AuthorizationEnforcer*. *Authorizatio-*

nEnforcer then sends *Subject* back to *Client*.

- **Security Properties** Authorization, if done properly, promotes separation of responsibility through access rights. It defines which resources an entity can access and with what access rights. That positively affects confidentiality, integrity and availability.
- **Known Uses** It is used as the basis for access control in many products, such as Windows, UNIX, MySQL (Schumacher et al., 2006).
- **Related Pattern** Authentication Enforcer (Steel et al., 2005) is required to authenticate users.
- **Source** Steel et al. (2005)

Authentication Enforcer

- **Intent** Verify the subject's identity
- **Alias** Authenticator (Fernandez and Sinibaldi, 2003), Authenticator (Blakley and Heath, 2004), Authenticator (F. Lee Brown et al., 1999), Authenticator (Schumacher et al., 2006), Agent Authenticator (Mouratidis et al., 2003), Message Authentication (Braga et al., 1998), Password Authentication (Kienzle and Elder, 2002), Remote Autehnticator/Authorizer (Fernandez and Warrier, 2003)
- **Problem** An attacker could try to impersonate a legitimate user to gain access to his resources. A way to prevent impostors is needed.
- **Solution** Have one component to receive interactions of a subject and verify the identity of the subject.
- **Participants** Client, AuthenticationEnforcer, RequestContext, Subject, UserStore
- **Interactions** The Client creates *RequestContext* containing user's credentials and invokes *AuthenicationEnforcer*'s authenticate, passing the RequestContext.

AuthenticationEnforcer retrieves the credentials and matches that to entries in *UserStore*. Upon a match, *AuthenticationEnforcer* then create a *Subject* for that user.

- **Security Properties** Authentication promotes confidentiality, integrity, and availability properties.
- **Known Uses** Most commercial operating system use passwords to authenticate their users. (Schumacher et al., 2006)
- **Related Pattern** Secure Pipe (Steel et al., 2005), Single Access Point (Yoder and Barcalow, 1997)
- **Source** Steel et al. (2005)

Secure Logger

- **Intent** Prevent an attacker from gathering potentially useful information about the system from system logs and to prevent an attacker from hiding their actions by editing system logs.
- **Alias** Secure Logger (Dougherty et al., 2009).
- **Problem** all application events must be securely logged for debugging and audit purposes.
- **Solution** use a secure logger to log messages in a secure manner.
- **Participants** Client, SecureLogger, LogManager, LogFactory, Logger and File.
- **Interactions** *Client* sends a log command to *SecureLogger*, together with the data to be logged. Upon receipt, the *secureLogger*, whose main responsibility is to collect the data, sends the data with a log command to the *LogManager*. The *LogManager* will request a new instance of *Logger* from *LogFactory*. The *Logger* is the component that logs data. It creates a new *File* and writes data into that file.
- **Security Properties** Confidentiality and integrity as only authorized user can

read from and write to the log files.

- **Known Uses** syslog-ng, SmartInspect, Windows XP Encrypting File System, TrueCrypt. (Dougherty et al., 2009)
- **Related Pattern** Secure Pipe (Steel et al., 2005), Encrypted Storage (Kienzle and Elder, 2002)
- **Source** Steel et al. (2005)

Encrypted Storage

- **Intent** Provides a second line of defense against the theft of data on system servers
- **Alias** Cryptographic Storage
- **Problem** Need an approach to protect sensitive data
- **Solution** The Encrypted Storage pattern encrypts the most critical user data before it is ever committed to disk.
- **Participants** Client, EncryptedStorage, Storage, EncryptDecrypt and Key
- **Interactions** *Client* sends a store command to *encryptedStorage*, together with the data to be stored. Upon receipt, the *encryptedStorage*, whose main responsibilities are to collect the data and to orchestrate the appropriate process, sends the data to *encryptDecrypt* with an encrypt command. *encryptDecrypt* then encrypts the data and return the encrypted data to *encryptedStorage*. *encryptedStorage* then sends the encrypted data to *storage* with a store command. *storage* then store the data. During the initial setup, *encryptedStorage* loads the value of the key to *encryptDecrypt* and keeps to itself the capability to the key.
- **Security Properties** Encrypted Storage increases confidentiality by ensuring that the data cannot be decrypted, even if it has been captured. Availability can be negatively affected if the encryption keys are lost.

- **Known Uses** The UNIX password file hashes each user's password and stores only the hashed form (Kienzle and Elder, 2002).
- **Related Pattern** Client Input Filters (Kienzle and Elder, 2002)
- **Source** Kienzle and Elder (2002)

Execution Domain

- **Intent** Define an execution environment, indicating explicitly all the resources in the domain.
- **Alias** Execution Domain (Fernandez, 2002)
- **Problem** Unauthorized components can destroy or modify information in files or databases which they are not supposed to.
- **Solution** Collect components into an execution domain.
- **Participants** multiple Domains, components in the system.
- **Interactions** define *Domain*s and assign *component*s to their respective *Domain*.
- **Security Properties** Confidentiality, integrity and availability are improved if the domains are set up correctly.
- **Known Uses** JVM defines restricted execution environments (Schumacher et al., 2006).
- **Related Pattern** Controlled process creator (Schumacher et al., 2006).
- **Source** Schumacher et al. (2006)

Appendix C

Security Patterns Catalog

C.1 Security Patterns

Table C.1: Security Pattern Catalog — Summary

Pattern Name	Intent	Source
3-Point Lgging	Log system events and system execution information	Dyson and Longshaw (2003)
3rd Party Communication	Understanding the risks of third party relationships	Romanosky (2001)
Access Control List	The access control list allows control access to objects by indicating which subjects can access an object and in what way. there is usually an acl associated with each object	Delessy et al. (2007)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Access Controller	Allow the agency to provide access to its resources according to its security policy	Mouratidis et al. (2003)
Account Lockout	Protects from password guessing attacks	Kienzle and Elder (2002)
Address Filter Firewall	To filter incoming and outgoing network traffic in a computer system based on network addresses	Fernandez et al. (2003)
Administrator Hierarchy	Define a hierarchy of system administrators with rights controlled using a role-based access control (rbac) model and assign rights according to their functions	Fernandez et al. (2006)
Administrator Object	Handles user-role assignment and delegates the administrative responsibilities	Kodituwakku et al. (2001)
Agency Guard	Provide a single, non-bypassable, point of access to the agency. the agency guard defines a structure that makes unauthorized access to the agency difficult	Mouratidis et al. (2003)
Agent Authenticator	Provide authentication services to the agency	Mouratidis et al. (2003)
Anonymity Set	Hide the data by mixing it with data from other sources	Hafiz (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Application Proxy Firewall	Inspect (and filter) incoming and outgoing network traffic based on the type of application they are accessing	Fernandez et al. (2003)
Assertion Builder	Structured and consistent approach to gathering security information (for example, saml assertions) about the authentication action performed on a subject	Steel et al. (2005)
Audit Interceptor	Intercept and audit service requests and responses for forensic purpose	Steel et al. (2005)
Authentication Enforcer	Verify that each request is from an authenticated entity	Steel et al. (2005)
Authenticator	Performs authentication of a requesting process before deciding access to distributed objects	Blakley and Heath (2004)
Authenticator	Performs authentication of a requesting process before deciding access to distributed objects	F. Lee Brown et al. (1999)
Authenticator	Verify that a user is who it says it is	Fernandez and Sinibaldi (2003)
Authenticator	This pattern addresses the problem of how to verify that a subject is who it says it is	Schumacher et al. (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Authoritative Source Of Data	Recognizing the correct source of data	Romanosky (2001)
Authorization	This pattern describes who is authorized to access specific resources in a system, in an environment in which we have resources whose access needs to be controlled	Schumacher et al. (2006)
Authorization Enforcer	Provides a centralized point for authorizing resources	Steel et al. (2005)
Authorization Pattern	Descibe who is authorized to access resources	Fernandez and Pan (2001)
Batched Routing	In a mix based system, collect the input data packets and when the collection reaches a threshold output all the data packets together	Hafiz (2006)
Capability	The capability pattern allows control access to objects by providing a credential or ticket to be given to a subject for accessing an object in a specific way. capabilities are given to the principal	Delessy et al. (2007)
Check Point	makes such an effective i&a and access control mechanism easy to deploy and evolve	Schumacher et al. (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Check Point	Propose a structure that checks incoming request. responsible for taking appropriate countermeasures in case of violations	Yoder and Barcalow (1997)
Checkpointed System	Structure a system so that its state can be recovered and restored to a known valid state in case a component fails	Blakley and Heath (2004)
Clear Sensitive Information	Ensures that sensitive information is cleared from reusable resources before the resource may be reused	Dougherty et al. (2009)
Client Data Storage	Encrypt data to allow secure storage on client	Kienzle and Elder (2002)
Client Input Filters	Protect application from data tampering on untrusted clients	Kienzle and Elder (2002)
Comparator-Checked Fault-Tolerant System	Structure a system so that an independent failure of one component will be detected quickly and will not cause a system failure	Blakley and Heath (2004)
Constant Length Padding	Add padding to data packets to make them of same length	Hafiz (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Constant Link Padding	Distribute data traffic equally among all the outgoing nodes from an anonymity preserving node	Hafiz (2006)
Container Manager Security	Provide a standard way to enforce authentication and authorization	Steel et al. (2005)
Continual Status Reporting	Report individual system element continuously, log some or all data for offline analysis	Dyson and Longshaw (2003)
Controlled Execution Environment	To control access to all operating system resources by processes, based on user, group, or role authorizations	Fernandez (2002)
Controlled Execution Environment	This pattern addresses how to control the execution environment	Schumacher et al. (2006)
Controlled Object Factory	This pattern addresses how to specify the rights of processes with respect to a new object	Schumacher et al. (2006)
Controlled Object Monitor	This pattern addresses how to control access by a process to an object. use a reference monitor to intercept access requests from processes	Schumacher et al. (2006)
Controlled Process Creator	This pattern addresses how to define and grant appropriate access rights for a new process	Schumacher et al. (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Controlled Virtual Address Space	This pattern addresses how to control access by processes to specific areas of their virtual address space (vas) according to a set of predefined access types	Schumacher et al. (2006)
Controlled-Object Creator	Create object for specific purposes and the rights to access other process	Fernandez and Sinibaldi (2003)
Controlled-Object Monitor	Define how to control access by subjects to objects	Fernandez and Sinibaldi (2003)
Controlled-Process Creator	Define and grant appropriate access rights for a new process	Fernandez and Sinibaldi (2003)
Cover Traffic	Keep a dummy traffic flow between anonymity preserving nodes to create a decoy for actual data traffic	Hafiz (2006)
Credential Delegation	Issue a special type of certificate (proxy certificate) signed by the original party (grantor) that con- firms that the holder of this certificate (grantee) is allowed to act on its behalf	Weiss (2006)
Credential Pattern	Provides secure portable means of recording authentication and authorization information for use in distributed system	Morrison and Fernandez (2006*a*)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Credential Tokenizer	Encapsulate a security token that can be used by different security infrastructure providers	Steel et al. (2005)
Cryptographic Metapattern	Define a generic software architecture to cryptography	Braga et al. (1998)
Defer To Kernel	Clearly separate functionality that requires elevated privileges from functionality that does not require elevated privileges and to take advantage of existing user verification functionality available at the kernel level	Dougherty et al. (2009)
Delayed Routing	Add random delays to the incoming data traffic of an anonymity preserving node to thwart the timing attacks	Hafiz (2006)
Demilitarized Zone	Separates the business functionality and information from the web servers that deliver it, and places the web servers in a secure area	Schumacher et al. (2006)
Directed Session	Ensure user cannot skip around within a series of activities	Kienzle and Elder (2002)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Distrustful Decomposition	Move separate functions into mutually untrusting programs, thereby reducing the attack surface of the individual programs that make up the system and functionality and data exposed to an attacker if one of the mutually untrusting programs is compromised	Dougherty et al. (2009)
Dynamic Service Management	Dynamically instrument fine-grained components to manager and monitor application	Steel et al. (2005)
Dynamically Adjustable Non-Functional Configuration	Adjusting the non-functional characteristics key parameters value without affecting system to run	Dyson and Longshaw (2003)
Encrypted Storage	Provide second line of defense against stealing of data by means of encryption	Kienzle and Elder (2002)
Error Detection/ Correction	Add redundancy to data to facilitate later detection of and recovery from errors	Blakley and Heath (2004)
Execution Domain	Define an execution environment for processes, indicating explicitly all the resources a process can use during its execution, as well as the type of access for the resources	Fernandez (2002)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Execution Domain	Define an execution environment for processes, indicating all the resources a process can use during its execution as well as the types of access for the resources	Schumacher et al. (2006)
Fail Securely	Designing systems to fail in a secure manner	Romanosky (2001)
Fault Container	Proposes the use of a wrapper that transforms a software component into its fault containing counterpart	Saridakis (2003)
File Authorization	Control access to files in an operating system. authorized users are the only ones that can use a file in specific ways	Fernandez (2002)
File Authorization	This pattern describes how to control access to files in an operating system	Schumacher et al. (2006)
File Authorization Pattern	Describe who is authorized to access to files	Fernandez and Pan (2001)
Firewall Pattern	Describe how access to internal networks can be restricted in general	Schumacher (2003)
Front Door	Provide entry point to system	Schumacher et al. (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Front Door	Provide a single log-in and session context	Sommerlad (2003)
Full Access With Error	Prevent users from performing illegal operations by failing them securely	Schumacher et al. (2006)
Full View W/ Errors	Prevent users from performing illegal operations by failing them securely	Yoder and Barcalow (1997)
Gooca	Generic software architecture for cryptographic applications	Braga et al. (1998)
Hidden Implementation	Limits attacker's ability to discern internal workings of system	Kienzle and Elder (2002)
Hidden Metadata	Hide the meta information associated with data content that reveal information about sensitive data content	Hafiz (2006)
Information Obscurity	Obscure data	Schumacher et al. (2006)
Information Secrecy	Keep the secrecy of information	Braga et al. (1998)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Informed Consent For Web-Based Transactions	Describes how websites can inform users whenever they intend to collect and use an individuals personal information	Romanosky et al. (2006)
Input Guard	Verify that every input fed to his component by the system conforms to the input for his component as described in the system specification	Saridakis (2003)
Input Validation	Correctly identify and validate all external inputs from untrusted data sources	Dougherty et al. (2009)
Integration Reverse Proxy	provides a homogenous view of a collection of servers, without leaking the physical distribution of the individual machines to end users	Schumacher et al. (2006)
Integration Reverse Proxy	Provides a homogenous view to a bunch of servers, without leaking the physical distribution to several machines to end users	Sommerlad (2003)
Intercepting Validator	Cleanse and validate data prior to its use within an application	Steel et al. (2005)
Interception Web Agent	Provide authentication and authorization external to the application by intercepting requests prior to the application	Steel et al. (2005)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Keep Session Data In The Client	Describes mechanisms to store session data on client side	Sørensen (2002)
Keep Session Data In The Server	Describes mechanisms to store session data on server side	Sørensen (2002)
Known Partners	Know who is interacting with the system	Schumacher et al. (2006)
Layered Encryption	Use a sender-initiated packet routing scheme and encrypt the data packets in multiple layers so that the in- termediaries only have access to a particular layer and use that information to route the packet to the next hop	Hafiz (2006)
Layered Security	Configuring multiple security checkpoints	Romanosky (2001)
Limited Access	This pattern guides a developer in presenting only the currently-available functions to a user, while hiding everything for which they lack permission	Schumacher et al. (2006)
Limited View	Prevent users from performing illegal operations by offering only valid operations	Yoder and Barcalow (1997)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Load Balancer	Spread a server application over several computers to handle more load than a single computer is capable of and to provide uninterrupted service to system users in the case of system break downs and scheduled system shutdowns	Sørensen (2002)
Low Hanging Fruit	Taking care of the quick wins	Romanosky (2001)
Masked Online Traffic	Provides solutions to help users protect their privacy by reducing the amount of information that they disclose while interacting online	Romanosky et al. (2006)
Message Inspector	Verify and validate the quality of message-level security mechanisms applied to XML web services	Steel et al. (2005)
Message Inspector Gateway	Provides a centralized entry point that encapsulates access to all target service endpoints of a web services provider and secures the incoming and outgoing XML traffic by securing the communication channels between the service endpoints	Steel et al. (2005)
Message Integrity	Avoid corruption of a message	Braga et al. (1998)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Minefield	Trick, block and detect attackers during break-in attempt	Kienzle and Elder (2002)
Minimal Information Asymmetry	Describes how you can protect your privacy by gathering more information about the parties whom you would like to transact online	Romanosky et al. (2006)
Morphed Representation	Change the representation of the data when it is passing through an anonymity providing node so that outgoing data cannot be linked with incoming data	Hafiz (2006)
Multilevel Security	this pattern describes how to categorize sensitive information and prevent its disclosure	Schumacher et al. (2006)
Multilevel Security Pattern	this pattern describes how to categorize sensitive information and prevent its disclosure	Fernandez and Pan (2001)
Network Address Blacklist	Keep track of network addresses that are sources of hacking attempts	Kienzle and Elder (2002)
Obfuscated Transfer Object	Protect critical data when passed within application and between tiers	Steel et al. (2005)
Operational Monitoring And Alerting	Report status of all system elements at an appropriate frequency	Dyson and Longshaw (2003)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Output Guard	Confines an error inside the component where that error occurred	Saridakis (2003)
Packet Filter	Restrict the ingoing and outgoing traffic at the border between the internal and the external network	Schumacher (2003)
Packet Filter Firewall	filters incoming and outgoing network traffic in a computer system based on packet inspection at the ip level	Schumacher et al. (2006)
Partitioned Application	Restrict dangerous privilege to a single component by means of isolation	Kienzle and Elder (2002)
Password Authentication	Protect against weak passwords and automated password-guessing attacks	Kienzle and Elder (2002)
Password Propagation	Provide alternative authentication that require authentication credentials be verified by system before access provided	Kienzle and Elder (2002)
Password Synchronizer	Synchronize the user passwords (or user credentials used for authentication and authorization) across different application systems using a programmatic interface	Steel et al. (2005)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Pathname Canonicalization	Ensure that all files read or written by a program are referred to by a valid path that does not contain any symbolic links or shortcuts, that is, a canonical path	Dougherty et al. (2009)
Policy	Isolate policy enforcement to a discrete component of an information system; ensure that policy enforcement activities are performed in the proper sequence	Blakley and Heath (2004)
Policy Delegate	Shield client from details of security services and control ther interactions	Steel et al. (2005)
Policy-Based Access Control	The policy-based access control pattern decides if a subject is authorized to access an object according to policies defined in a central policy repository	Delessy et al. (2007)
Privacy-Aware Network Client	Provides a way to make a user of a network site aware of the privacy policies followed by that site	Sadicoff et al. (2005)
Privilege Separation	Reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the program	Dougherty et al. (2009)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Privillege-Limited Role	Implement role classes based on privileges of the corresponding role	Kodituwakku et al. (2001)
Protected System	Structure a system so that all access by clients to resources is mediated by a guard which enforces a security policy	Blakley and Heath (2004)
Protection Against Cookies	Provides countermeasures against the misuse of cookies in the WWW	Schumacher (2002)
Protection Reverse Proxy	Protects the server software at the level of the application protocol	Schumacher et al. (2006)
Proxy-Based Firewall	Restrict the ingoing and outgoing traffic at the border between the internal and the external network with a firewall	Schumacher (2003)
Proxy-Based Firewall	this pattern interposes a proxy between the request and the access, and applies controls through this proxy	Schumacher et al. (2006)
Pseudonymous Email	Protect against unforeseen ramifications of e-mail messages	Schumacher (2002)
RBAC Pattern	Introduces roles to access protected information objects on behalf of users and introduces distinct roles to administer users and roles	Fernandez and Pan (2001)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Reference Monitor	Enforce authorizations when a process requests resources	Fernandez (2002)
Reference Monitor	Make it possible that all authorizations are fulfilled when a process requires resources	Schumacher et al. (2006)
Remote Authenticator/Authorizer	Provide authentication and authorization for accessing shared resources	Fernandez and Warrier (2003)
Replicated System	Structure a system which allows provision of service from multiple points of presence, and recovery in case of failure of one or more components or links	Blakley and Heath (2004)
Resource Acquisition Is Initialization (RAII)	Ensure that system resources are properly allocated and deallocated under all possible program execution paths	Dougherty et al. (2009)
Risk Assessment And Management	Understanding the relative value of information and protecting it accordingly	Romanosky (2001)
Risk Determination	Evaluate and prioritize the risks to its assets	Schumacher et al. (2006)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Role Rights Definition	This pattern provides a precise way, based on use cases, of assigning rights to roles to implement a least-privilege policy	Schumacher et al. (2006)
Role Validator	Validates user-specified roles	Kodituwakku et al. (2001)
Role-Based Access Control	This pattern describes how to assign rights based on the functions or tasks of people in an environment in which control of access to computing resources is required and where there is a large number of users, information types, or a large variety of resources	Schumacher et al. (2006)
Role-Based-Access	Introduces roles to access protected information objects on behalf of users and introduces distinct roles to administer users and roles	Kodituwakku et al. (2001)
Role-Hierarchies	Forms role hierarchies and implements the different access privileges of different role	Kodituwakku et al. (2001)
Roles	Improve maintainability of privileges	Yoder and Barcalow (1997)
Sandbox	Allow the agency to execute non-authorised agents in a secure manner	Mouratidis et al. (2003)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Secrecy With Authentication	Prove the authenticity of a secret	Braga et al. (1998)
Secrecy With Integrity	Keep the integrity of a secret	Braga et al. (1998)
Secrecy With Signature	Prove the authorship of a secret	Braga et al. (1998)
Secrecy With Signature With Appendix	Separate secret from signature	Braga et al. (1998)
Secure Access Layer	Provide a secure gateway for communicating in and out of a program	Yoder and Barcalow (1997)
Secure Assertion	Spread application-specific checks throughout the system	Kienzle and Elder (2002)
Secure Base Action	Coordinate security components and provide web tier components with a central access point for administering security related functionality	Steel et al. (2005)
Secure Broker	Amends broker to provide secure interactions between distributed components	Morrison and Fernandez (2006*b*)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Secure Builder Factory	Separate the security dependent rules involved in creating a complex object from the basic steps involved in actually creating the object	Dougherty et al. (2009)
Secure Chain Of Responsibility	Decouple the logic that determines user/environment-trust dependent functionality from the portion of the application requesting the functionality, simplify the logic that determines user/environment-trust dependent functionality, and make it relatively easy to dynamically change the user/environment-trust dependent functionality.	Dougherty et al. (2009)
Secure Channels	for sensitive communication across a public network, create encrypted secure channels to ensure that data remains confidential in transit	Schumacher et al. (2006)
Secure Communcation	Ensure that mutual security policy objectives are met when there is a need for two parties to communicate in the presence of threats	Blakley and Heath (2004)
Secure Directory	Ensure that an attacker cannot manipulate the files used by a program during the execution of the program	Dougherty et al. (2009)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Secure Factory	Separate the security dependent logic involved in creating or selecting an object from the basic functionality of the created or selected object	Dougherty et al. (2009)
Secure Logger	Prevent an attacker from gathering potentially useful information about the system from system logs and to prevent an attacker from hiding their actions by editing system logs	Dougherty et al. (2009)
Secure Logger	Log messages in a secure manner so that they cannot be easily altered or deleted and so that events cannot be lost	Steel et al. (2005)
Secure Message Router	Securely communicate with multiple partner endpoints using message-level security and identity-federation mechanisms	Steel et al. (2005)
Secure Pipe	Guarantee the integrity and privacy of data sent over the wire	Steel et al. (2005)
Secure Process/Thread	Make sure a process does not interfere with other processes	Fernandez et al. (2006)
Secure Proxy	Define the relationship between the guards of two instances of protected system in the case when one instance is entirely contained within the other	Blakley and Heath (2004)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Secure Service Facade	Provide a gateway governing security on client requests	Steel et al. (2005)
Secure Service Proxy	Provide authentication and authorization externally by intercepting requests for security checks	Steel et al. (2005)
Secure Session Manager	defines how to create a secure session by capturing session information	Steel et al. (2005)
Secure Session Object	Facilitate distributed access of security context and sessions	Steel et al. (2005)
Secure State Machine	Allow a clear separation between security mechanisms and user-level functionality by implementing the security and user-level functionality as two separate state machines	Dougherty et al. (2009)
Secure Strategy Factory	Provide an easy to use and modify method for selecting the appropriate strategy object (an object implementing the strategy pattern) for performing a task based on the security credentials of a user or environment	Dougherty et al. (2009)
Secure Visitor	Prevent unauthorized access to nodes in the data structure	Dougherty et al. (2009)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Security Association	Define a structure which provides each participant in a secure communication with the information it will use to protect messages to be transmitted to the other party, and with the information which it will use to understand and verify the protection applied to messagesreceived from the other party	Blakley and Heath (2004)
Security Context	Provide a container for security attributes and data relating to a particular execution context, process, operation or action	Blakley and Heath (2004)
Security Reverse Proxy	Protects your server from attacks on network and application protocol levels	Sommerlad (2003)
Security Session	a unique reference to the session object is made available, instead of passing all access rights or re-authenticating a user repeatedly	Schumacher et al. (2006)
Sender Authentication	Avoid refusal of a message	Braga et al. (1998)
Server Sandbox	Contain damage resulting from undiscovered bug in server	Kienzle and Elder (2002)
Session	Describes a wide spread way of implementing state-fullness in a multi-user system	Sørensen (2002)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Session	Provide different parts of a system with global information about a user	Yoder and Barcalow (1997)
Session Failover	Describes a way to keep session related data available to the users of the system, even in the case of system shutdown or breakdowns	Sørensen (2002)
Session Scope	Storing session specific data on the server and accessing it from code in a way that minimizes the risk of mixing up data from several sessions	Sørensen (2002)
Session Timeout	Guard against session spefic data stored in a server growing to fill up all available memory and disk	Sørensen (2002)
Signature	Proivde the authorship of a message	Braga et al. (1998)
Signature With Appendix	Separate message from signature	Braga et al. (1998)
Single Access Point	grants or denies entry to the system after checking the client requiring access	Schumacher et al. (2006)
Single Access Point	Define a single entry point for the system	Yoder and Barcalow (1997)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Single Session	Creates a session with a validated role	Kodituwakku et al. (2001)
Single Sign On	Provide mechanism to avoid re-authentication of user after successful authentication	Kienzle and Elder (2002)
Single Sign-On Delegator	Encapsulate access to identity management and single sign-on functionalities, allowing independent evolution of loosely coupled identity management services while providing system availability	Steel et al. (2005)
Standby	Structure a system so that the service provided by one component can be resumed from a different component	Blakley and Heath (2004)
Stateful Firewall	filters incoming and outgoing network traffic in a computer system based on state information derived from past communications	Schumacher et al. (2006)
Subject Descriptor	Provide access to security-relevant attributes of an entity on whose behalf operations are to be performed	Blakley and Heath (2004)
System Overview	Monitor all the interfaces to each of the system elements individually	Dyson and Longshaw (2003)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
The Object Filter And Access Control Framework	This framework combines the functions of authentication, access control, and data filtering in a distributed environment	Hays et al. (2000)
The Role -Based Security Assertion Coordinator	The role -based security assertion coordinator pattern allows seamless exchange of security data among organizations in a distributed environment in order to coordinate role-based access control to protected resources	Fernandez (2004)
The Security Provider	Leveraging the power of a common security service across multiple applications	Romanosky (2001)
The XML Firewall Pattern	To filter XML messages to/from user-defined applications, based on the business access control policies and the content of the message	Fernandez (2004)
Trusted Proxy	Provide safe interface by constraining access to protected resources	Kienzle and Elder (2002)
Validated Transaction	Puts all security relevant validation into one transaction	Kienzle and Elder (2002)
Virtual Address Space Access Control	To control access by processes to specific areas of their virtual address space (vas) according to a set of predefined access types	Fernandez (2002)

Continued on next page

Table C.1 – continued from previous page

Pattern Name	Intent	Source
Virtual Address Space Structure Selection	Emphasize isolation for virtual address space	Fernandez et al. (2006)
White Hats, Hack Thyself	Testing your own security by trying to defeat it	Romanosky (2001)
WSPL	WSPL enables an organization to represent access control policies for its web services in a standard manner. it also enables a web services consumer to express its requirements in a standard manner	Delessy and Fernandez (2005)
XACML Access Control Evaluation	This pattern decides if a request is authorized to access a resource according to policies defined by the XACML authorization pattern	Delessy and Fernandez (2005)
XACML Authorization	Enables an organization to represent authorization rules in a standard manner	Delessy and Fernandez (2005)
	Total number of patterns	200

C.2 Non-Design Security Patterns

Table C.2: Security Pattern Catalog — Non Design Patterns

Pattern Name	Source
Access Control Requirements	Schumacher et al. (2006)
Account Category	Riehle et al. (2002)
Actor and Role Lifecycle Pattern	Elsinga and Hofman (2002)
Actor-Based Access Rights	Elsinga and Hofman (2002)
Address Book	Lehtonen and Pärssinen (2001)
Alice and Friends	Lehtonen and Pärssinen (2001)
Asset Valuation	Schumacher et al. (2006)
Audit Requirements	Schumacher et al. (2006)
Audit Trails and Logging Requirements	Schumacher et al. (2006)
Automated I&A Design Alternatives	Schumacher et al. (2006)
Biometrics Design Alternatives	Schumacher et al. (2006)
Build The Server From Ground Up	Kienzle and Elder (2002)
Choose The Right Stuff	Kienzle and Elder (2002)
Codebook	Riehle et al. (2002)
Community of Nodes	Pärssinen and Turunen (2002)

Continued on next page

Table C.2 – continued from previous page

Pattern Name	Source
Content of a Message for Humans	Pärssinen and Turunen (2002)
Content of a Message for Machines	Pärssinen and Turunen (2002)
Conversation Between Nodes	Pärssinen and Turunen (2002)
Dictionary Word	Riehle et al. (2002)
Document The Security Goals	Kienzle and Elder (2002)
Document The Server Configuration	Kienzle and Elder (2002)
Elements of a Node	Pärssinen and Turunen (2002)
Enroll by Validating Out of Band	Kienzle and Elder (2002)
Enroll Using Third-Party Validation	Kienzle and Elder (2002)
Enroll with a Pre-Existing Shared Secret	Kienzle and Elder (2002)
Enroll without Validating	Kienzle and Elder (2002)
Enterprise Partner Communication	Schumacher et al. (2006)
Enterprise Security Approaches	Schumacher et al. (2006)
Enterprise Security Services	Schumacher et al. (2006)
Face-to-Face	Lehtonen and Pärssinen (2001)
From Service to Protocol	Pärssinen and Turunen (2002)
I&A Requirements	Schumacher et al. (2006)

Continued on next page

Table C.2 – continued from previous page

Pattern Name	Source
Interfaces of an Entity	Pärssinen and Turunen (2002)
Intrusion Detection Requirements	Schumacher et al. (2006)
Keep It Secret	Riehle et al. (2002)
Key in The Pocket	Lehtonen and Pärssinen (2001)
Lay It Open	Riehle et al. (2002)
Log for Audit	Kienzle and Elder (2002)
Master Account File	Riehle et al. (2002)
Means to Communicate	Pärssinen and Turunen (2002)
Message Exchange	Pärssinen and Turunen (2002)
Message Identification	Pärssinen and Turunen (2002)
Message Transfer Syntax	Pärssinen and Turunen (2002)
Message Versioning	Pärssinen and Turunen (2002)
Needs to Communicate	Pärssinen and Turunen (2002)
Non-Repudiation Requirements	Schumacher et al. (2006)
Parameter Container	Pärssinen and Turunen (2002)
Password Algorithm	Riehle et al. (2002)
Password Design and Use	Schumacher et al. (2006)

Continued on next page

Table C.2 – continued from previous page

Pattern Name	Source
Password Externalization	Riehle et al. (2002)
Password Hint	Riehle et al. (2002)
Password Lock Box	Riehle et al. (2002)
Password Salt	Riehle et al. (2002)
Patch Proactively	Kienzle and Elder (2002)
Piggy Packing	Pärssinen and Turunen (2002)
Read Team The Design	Kienzle and Elder (2002)
Seal Ring Engraver	Lehtonen and Pärssinen (2001)
Sealed and Signed Envelope	Lehtonen and Pärssinen (2001)
Sealed Envelope	Lehtonen and Pärssinen (2001)
Security Accounting Requirements	Schumacher et al. (2006)
Security Context	Riehle et al. (2002)
Security Needs Identifcation for Enterprise Assets	Schumacher et al. (2006)
Share Responsibility for Security	Kienzle and Elder (2002)
Signed Envelope	Lehtonen and Pärssinen (2001)
Singleton Password	Riehle et al. (2002)
Stay Current and Ahead	Riehle et al. (2002)

Continued on next page

Table C.2 – continued from previous page

Pattern Name	**Source**
Tail Extension	Pärssinen and Turunen (2002)
Test on a Staging Server	Kienzle and Elder (2002)
The Forged Seal Ring	Lehtonen and Pärssinen (2001)
The Goal of Security	Elsinga and Hofman (2003)
The Nature of Safeguards	Elsinga and Hofman (2003)
The Real Thing	Lehtonen and Pärssinen (2001)
There is Somebody Eavesdropping	Lehtonen and Pärssinen (2001)
Threat Assessment	Schumacher et al. (2006)
Transmission Media	Pärssinen and Turunen (2002)
Two Roles of Nodes	Pärssinen and Turunen (2002)
Typing Rhythm	Riehle et al. (2002)
Unusual Variation	Riehle et al. (2002)
Vulnerability Assessment	Schumacher et al. (2006)
Total number of patterns	79