Multi-Representational Security Modeling and Analysis

by

Eunsuk Kang

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
January 29, 2016

Signature redacted

Certified by ....
Daniel Jackson
Professor
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by ....
Leslie A. Kolodziejski
Professor
Department of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Multi-Representational Security Modeling and Analysis

by
Eunsuk Kang

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Many security attacks arise from unanticipated behaviors that are inadvertently introduced by the system designer at various stages of the development. This thesis proposes a *multi-representational* approach to security modeling and analysis, where models capturing distinct (but possibly overlapping) views of a system are automatically composed in order to enable an end-to-end analysis. This approach allows the designer to incrementally explore the impact of design decisions on security, and discover attacks that span multiple layers of the system. The thesis also introduces *Poirot*, a prototype implementation of the approach, and reports on the application of Poirot to detect previously unknown security flaws in publicly deployed systems.

Thesis Supervisor: Daniel Jackson
Title: Professor
Department of Electrical Engineering and Computer Science

Acknowledgments

I feel incredibly fortunate to have been under Daniel's guidance during my time at MIT. As a young, foolish graduate student who didn't know any better, I was particularly apt at digging myself into a hole on numerous occasions, often against his warnings. But no matter how much trouble I got myself into, he has always been extremely generous and supportive of my decisions. He has never shied away from giving me constructive criticism when needed, and always encouraged me to pursue a broad vision and freely explore my own research interests. I will miss being imbued with high energy and enthusiasm every time I walked into a meeting with him.

Ethan has been a great mentor and colleague to me ever since my internship at Microsoft Research. Being on the industrial side of research, he has encouraged me to look at things from different perspectives, always to my benefits, and his continual, fearless effort to bridge the gap between theory and practice has been simply awe-inspiring.

Rob's desk is located in an area right outside my office—one of the loudest places in the entire lab during the past several years—and yet he has been extremely patient and tolerant of our ruckus. Being near his research group has always been a refreshing experience, and I am grateful for all his advice and suggestions on improving my thesis.

My mentors at Waterloo—Jo, Nancy, and Mark—are the ones who got me interested in the wonderful (and challenging) world of formal methods. I could simply not be where I am today without their continual guidance and support over the years.

I am extremely grateful to the members of the Software Design Group: Felix, Greg, Jonathan, Ivane, Matt, Aleks, Joe, Derek, Santiago, Rob, Rishabh, Emina, and Kuat. Every one of them has had a significant influence on my ideas and perspectives, which I will continue to carry with me for many years to come. Special thanks to Maria for being tolerant of all the hassle that I've caused around her desk over the years, and for being a wonderful, caring person to all of us.

It would be a huge understatement to say that my life at MIT has been fun and enjoyable thanks to the members of the MIT Lambda group: It has been, at times, dangerous, stimulating, and bordering on madness. I feel extremely lucky to have been part of some of the most unique, wonderful people I've ever seen assembled: James, Michal, Marcie, Sachi, Ivane, Aleks, Sasa, Joe, Rishabh, Santiago, Eugene, Jean, and Kuat.

I owe huge thanks to my friends at MIT and back in Canada for their continual support during both my ups and downs: Andrew, Alvin, Steve, James, Michal, Edward, Max, Philip, Neha, Sachi, Shachar, Harshad, Bryce, Theresa, Fred, Ivane, Florbela, Joseph, Aleks, Sasa, Ashley, Joe, Nadia, Evan, Santiago, Oshani, Gabi, Rishabh, Rohit, Marija, Vanessa, Dilini, Eugene, Jean, Jones, and Dave.

I would like to thank my collaborators for everything they have taught me: Sridhar Adepu, Hamid Bagheri, Alcino Cunha, Christian Estler, Vijay Ganesh, Xavier Gillard, Jianye Hao, Rajeev Joshi, Sun Jun, Aleksandar Milicevic, Joseph Near, Derek Rayside, and Liu Yang.

I would like to give special thanks to Jackie for her unconditional support and encouragement throughout the years.

Finally, I dedicate this thesis to my amazing sister and parents. As always, no words are sufficient to express my gratitude to them.

Contents

List of Figures

Chapter 1

Introduction

When a system suffers a catastrophic security failure, the blame is often placed on the developers of the system: They must have failed to invest enough effort into security, or perhaps they are simply incompetent.

Recent evidences suggest that the story may not be so simple. According to an industry survey [85], the amount of development cost spent on security is higher than ever, and yet the number of vulnerabilities in software products continues to grow significantly each year. Some of the systems that were *proven* secure using the most rigorous techniques available (e.g., formal verification) have been shown to be vulnerable to relatively simple attacks [47, 74, 58]. Contrary to what one may expect, many of the recent failures are not due to a novel attack, but caused by a recurrence or slight variation of previously known vulnerabilities.

This thesis aims to provide a different approach to understanding security failures. It begins with an observation that may be surprising but not novel: What makes security particularly challenging is also a fundamental enabler of the construction of complex systems: *abstraction*.

1.1 Risk of Abstraction

Abstraction is a technique for managing the complexity of a system by suppressing certain details from its description. The technique is applied in every stage of development, starting from the early conceptual design and implementation to testing and documentation. Abstraction itself can be achieved using various methods, including separation of concerns [27], information hiding [68], layering [78], and encapsulation [51].

One of the key benefits of abstraction is that, by removing irrelevant details, it can be used to simplify reasoning about properties of a system. Imagine the task of designing an online store system. When reasoning about the interaction between a customer and a shopping cart, the designer would ignore details about other aspects of the system, such as payment processing and product delivery. Low-level design decisions, such as the choice of the underlying computing platform and data structures, would be deferred until the business logic has been fully laid out and scrutinized.

However, in domains where security is a paramount concern, abstraction can be a double-edged sword. A key observation, noted since early days of security [49], is that

many security attacks arise due to discrepancies between the designer's view of the system and that of an attacker. While the designer tends to operate on a single abstraction at a time, the attacker is not bound by such restrictions, and may exploit weaknesses across multiple layers or aspects of a system. The details that are deliberately ignored by the designer may be exactly what allow the attacker to undermine the security of the system.

Let us explore some concrete examples that demonstrate this type of risk.

1.1.1 Insecure Protocol Deployment

OAuth is one of the most widely used authorization protocols on the web [40]. The protocol is used to perform *third-party authorization*; that is, it allows an application to access resources from another service provider, pending the approval of the end user that owns those resources. For example, a third-party merchant application may use OAuth to access a user's billing information on Amazon, after the user indicates an approval by partaking in the protocol.

Due to its popularity and importance, OAuth has been subjected to careful scrutiny [40] and rigorous analysis, including formal verification [20, 67, 89]. Despite the amount of attention that it has received, however, a majority of web applications relying on OAuth have been shown to be insecure, allowing an attacker to bypass the protocol and access sensitive resources without the user's approval [79]. In many of these applications, the vulnerability was caused by a logical flaw, not simple programming errors such as buffer overflow or missing input validation.

Most of the security analysis on OAuth was performed on the *protocol* specification. However, depending on the underlying *platform* on which the protocol is deployed, it may become susceptible to a wide range of attacks that cannot even be represented at the protocol level. For example, one of the attacks relies on a cross-site request forgery (CSRF), which exploits the way standard web browsers handle cookies. But the protocol is designed to be platform-independent, and does not talk about browsers; it is not surprising that an analysis at this level would not be able to capture this attack[1]!

A number of protocols besides OAuth have suffered from a similar type of problem [63, 60]. In general, a secure protocol does not necessarily lead to a secure system. The core of the issue, we hypothesize, is the mismatch between the protocol designer's view of the system and that of an attacker. The designer rightfully focuses on high-level interactions between protocol participants, and omits discussions about how these participants are to be realized in a concrete system. But it may be one of these details that allows the attacker to render the protocol insecure.

1.1.2 Side Channel Attacks

Consider an online store as an example. When a user wishes to perform a stateful operation (such as adding an item to a shopping cart), the server authenticates the user by requiring a secret credential to be presented along with the request. The server checks the provided credential against the existing record on its database, and only if they match, it considers the request to have originated from the valid user.

[1]The RFC for OAuth [40] does discuss the potential risk of CSRF against OAuth implementations, but as the study [79] suggests, this warning seems to have gone unheeded by most developers.

One of the design decisions faced by the store developer is the choice of data structure for encoding credentials inside the server application. A typical option would be to encode a credential as an array of bytes; then, the authentication process would include a step where two arrays are compared for their content. A straightforward implementation of the comparison method compares one byte at a time, and as soon as it finds a pair of bytes that differ, returns `false` as the result. For example, consider the following snippet from the `Arrays.equals` method in the Java standard library [31]:

```
public static boolean equals(byte[] a, byte[] a2) {
    ...
    for (int i=0; i < a.length; i++)
        if (a[i] != a2[i])
            return false;

    return true;
}
```

Unfortunately, this seemingly innocuous implementation could reveal information on the user's credential to the attacker. Note that the amount of time it takes for this method to terminate depends on the number of matching starting bytes in the input arrays; greater the length of the matching prefix, longer it takes for the method to return `false`. By exploiting this property and systematically enumerating candidate credentials, the attacker could correctly deduce a user's credential. One may expect that the time difference should negligible enough to render this attack infeasible; however, this attack has been successfully carried out against real applications on the web [33].

This attack is an example of *side-channel attack*. Our hypothesis is that many side channel attacks can be phrased as a violation of abstraction. Most developers would pay little attention to the precise timing characteristics of a program; it typically has little impact on the overall functionality! This example demonstrates that simplicity of design is sometimes at odds with security; achieving the latter often results in additional complexity to the design of the system.

1.1.3 Feature Interaction

Let us look at a different type of security failure, where one or more independent *features* of a system interact in an unexpected manner that leads to a security violation.

A typical telecommunication company offers a wide range of services to its users, in addition to the basic ability to make and receive a call [45]. For example, a user may wish to keep her number private, and subscribe to a service called *calling line identity restriction* (CLIP), which blocks the caller's number from appearing on the callee's phone. The company may also offer a service called *automatic recall* (AR), which allows a user to automatically return the most recent incoming call, without having access to the number of the caller. These two features are complementary; CLIP ensures the confidentiality of one's number, and AR allows communication between two users without comprising the guarantee provided by CLIP.

Let us consider a third type of feature called *itemized billing* (ITM). This service provides a detailed bill with a list of all numbers that have been dialed by a user over the billing period. Extra caution must be taken when a user subscribes to both AR and ITM; unless explicitly sanitized, the bill may include the numbers from users who subscribe to CLIP, which is a clear violation of confidentiality!

The issue at heart is that these features are developed independently from one another; by focusing on one feature at a time, the designer ignores parts of the system that are irrelevant to the feature. It is unlikely that, for example, when devising the AR feature, the designer would be concerned with the information displayed on a user's bill. But eventually, these features need to be analyzed together for subtle interaction that may lead to an undesirable outcome. While this example with CLIP may seem simple, consider that the number of available features may exceed a hundred; predicting all unwanted interaction is a non-trivial task.

1.2 The Designer's Dilemma

The examples that we have discussed so far demonstrate the inherent tension between abstraction and security. On one hand, the only way we can manage the sheer complexity of a modern computer system is simplifying its description down to the bare essence. On the other, we have seen that seemingly irrelevant details can be exploited to compromise the security of the entire system. How does the designer determine which details are relevant for security, and which can be safely ignored?

As an attempt to answer this question, let us consider a different kind of domain: safety engineering in aviation. A commercial airliner is an enormously complex system, operating in an environment with numerous hazards, and yet has shown to be remarkably safe. But this was not always the case; in early years of the aviation industry, accidents were much more frequent[2]. For example, one of the common causes of the early accidents was a *bird strike*—a collision with a bird or wildlife resulting in aircraft damage. Bird strikes have been reported since the early 20th century, but it was a fatal crash of a civil airliner in 1960 that initiated a FAA mandate for a robust design of jet engines that are capable of withstanding bird strikes [6].

A comparison between aviation safety and computer security should be approached with caution, as they deal with very different kinds of system. However, we believe that some of the lessons learned from designing safe aircraft are also applicable to software systems—in particular, the practice of **codifying and reusing domain knowledge** gained from previous failures.

This approach *has* been adopted in certain domains, albeit in a rather ad hoc, sporadic manner. For example, in web security, a wealth of knowledge has been accumulated over the past decade about common vulnerabilities and mitigations. However, the information is scattered across numerous places in various forms—informal articles, research papers, checklists, and books, just to name a few. It requires a considerable amount of effort for an average web developer to consolidate and apply this knowledge to a specific system. The OWASP database, one of the most reputable sources on web security, lists more than 70 different types of attack on web applications [87]; how does one decide which out of these are actually applicable to a system being designed?

[2]In the United States, there were roughly 78 accidents per 100,000 flight hours in 1946, compared to 7 in 2009 [66].

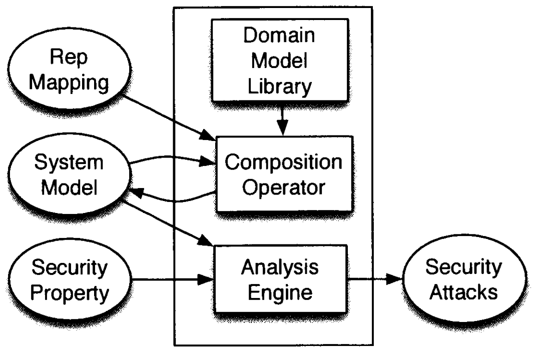


Figure 1-1: Overview of the proposed framework.

1.3 Proposed Approach

This thesis proposes a framework for designing secure systems through systematic reuse of domain knowledge. Here, the term *domain knowledge* has a rather broad meaning, in that it represents any *partial description* of a system (or a class of systems); it could, for example, describe a generic protocol (e.g., OAuth), a mechanism (how a browser stores and transmits cookies), a security attack (CSRF), or a data representation (an array of bytes for encoding a credential).

The overview of our framework is shown in Figure 1-1. It accepts three types of input from the user: a model of the system, a desired security property, and an optional *representation mapping*, which relates parts of the model to existing concepts in the *domain model library*. The *composition operator* constructs a new, elaborated model of the system by composing the given model with relevant domain models from the library, as specified in the mapping. Then, the *analysis engine* performs an automated analysis to generate potential scenarios that demonstrate how the property could be violated by an attacker. The library contains generic, reusable descriptions of systems (such as web applications and network systems), and can be easily extended with additional models.

The framework was designed to achieve the following specific goals:

- **Incremental analysis**: The framework is intended to support an **incremental** analysis of a system. Instead of having to come up with a complete model of the system at once, the designer may begin with a simple, abstract model, and incrementally elaborate parts of it. This allows the analysis task to be carried out over **multiple phases**. For example, the designer may first analyze an abstract OAuth model for protocol-specific flaws; elaborate the model by describing how the protocol is deployed on a HTTP server; and then perform an additional analysis to discover HTTP-specific attacks on OAuth. This approach is especially useful during early stages of development, where some of the design decisions may be unknown and still being explored.

- **Reasoning across multiple abstractions**: The underlying specification mechanism allows a system to be described in **multiple representations**, each corresponding to a particular abstraction of the system. This, in turn, enables an analysis for discovering attacks that exploit the behavior of the system across multiple abstraction layers.

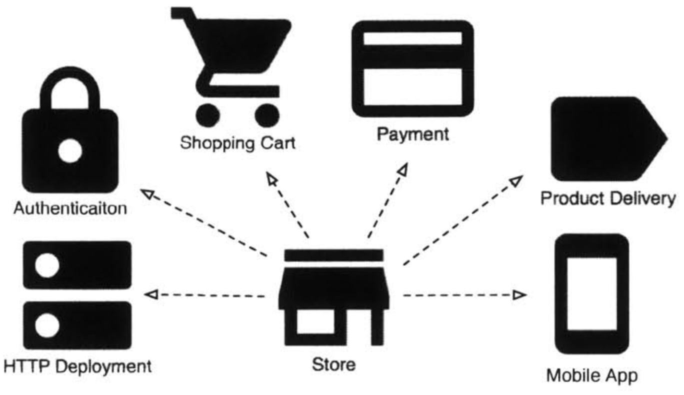


Figure 1-2: Different views of an online store system. An edge represents a projection from the system to one of its views.

For example, an analysis may discover an attack that relies on a subtle interplay between the OAuth and HTTP protocols, besides those that are purely OAuth- or HTTP-specific.

- **Modular reuse of domain knowledge**: By allowing models to be developed independently and later composed for analysis, the framework facilitates reuse of domain knowledge. A model that describes the behavior of a standard browser, for example, needs to be constructed only **once** by a security expert, and can be reused for analysis of **multiple** applications. Furthermore, the designer can leverage the domain knowledge without being exposed to all of its underlying details. For example, a web developer should be able to perform an analysis to discover potential attacks on the system, without understanding the complex inner workings of a browser.

1.3.1 Problem: Composing Mismatched Models

As a key requirement, this framework was designed so that a piece of domain knowledge could be encapsulated as a standalone model, and readily composed with other models, much like modules in programming languages. In traditional model-based frameworks [19, 25, 35], composition involves joining two models at a common, shared interface with certain synchronization primitives. In our framework, however, we face a rather different kind of model composition problem.

Two models may be developed independently, each with its own distinct vocabulary, but parts of them may *overlap*, in that they conceptually describe the same entity in the real world. When reasoning about the business logic, it is useful to treat an online store as an abstract entity, free of details about its underlying platform. But eventually, the store will be realized as a specialized type of web server that provides store-related services via HTTP requests. So, depending on the development task at hand, the same store entity in the world may be described using distinct concepts in different models—as the abstract store in one model, and as a HTTP server in another.

A different way to think about the composition problem is to treat each model as a *projection* of the system onto one particular aspect or abstraction layer. Figure 1-2 shows an online store system and some of its possible projections; one of them describes the shopping cart logic, while another depicts how the customers are authenticated. While these projections capture different aspects of the system, they are likely to overlap in some parts. For example, both the shopping cart and payment projections must include concepts that correspond to customers and items, although they may be represented differently.

We start with an assumption that no model of a system is perfect, and thus, it may never be possible to obtain a complete description from which projections can be constructed (e.g., the one in the center of Figure 1-2). Instead, we posit that the designer creates and switches between different projections at various points throughout the development. Since certain security attacks exploit details across multiple projections, these eventually need to be brought together for an end-to-end analysis.

In this thesis, we introduce a simple, general concept called the *representation mapping*, which expresses relationships between different elements of a system. In Chapter 2, we will discuss how the representation mapping is used as the basic operator for composing models, and the role that it plays in the context of security analysis.

1.3.2 Scope of the Framework

Computer security is a large, active area of research, and a number of techniques and tools have been developed over the past to address different types of vulnerabilities and attacks.

Our framework is intended to be used for **design analysis**, helping developers (1) discover potential attacks due to a **logical flaw** or an **unanticipated interaction** between different parts of the system, and (2) explore **implications of design alternatives** on the security of the system. Types of question that one may explore using our framework include:

- What are different ways in which an attacker might be able to access users' sensitive data on the store?
- Which method of transmitting user sessions is more secure: cookies or URL query parameters?
- How do I securely integrate the store with a third-party payment API?

On the other hand, the framework is not designed for finding specific vulnerabilities or bugs in the implementation of a system. It is, for example, not suited for detecting occurrences of buffer overflow or SQL injection on a piece of code. These implementation-level issues are just as crucial in security; fortunately, a number of tools have been built and optimized for analyzing such issues [75, 34].

1.4 Prototype Implementation and Evaluation

To demonstrate the feasibility of our approach, we have built a prototype tool called *Poirot*. The tool implements all of the major components from Figure 1-1, with a domain-specific language for constructing system models, and an analysis backend that uses the Alloy Analyzer [42] to generate potential security violations.

For evaluation, we focused on analyzing systems from one particular domain: web applications. For this purpose, we populated the library inside Poirot with a number of models that describe different aspects of the web, including communication protocols, encryption, browser features, and authentication mechanisms. We then applied Poirot to model and analyze a number of systems, two of which are described in this thesis:

- **HandMe.In**: A web-based system for tracking personal physical properties, and
- **IFTTT**: An application that allows end users to compose and automate a pair of web services.

Through our analysis, we identified a number of security flaws in the design of these systems, some of which could be used to carry out previously unknown attacks. More details about Poirot and its applications are discussed in Chapters 5 and 6.

1.5 Summary of Thesis Contributions

This thesis makes the following contributions:

- A modeling and analysis framework that leverages domain knowledge to detect potential security attacks across multiple system abstractions.
- A general mechanism for composing independent, possibly overlapping models by relating different representations of system entities.
- A prototype tool, called Poirot, and case studies describing its application to the analysis of realistic systems.

1.6 Thesis Outline

The rest of the thesis is structured as follows. Chapter 2 describes our underlying formalism for specifying a system, and a new composition method based on the notion of the *representation mapping*. Chapter 3 introduces a model for describing dataflow throughout a system, and shows how a representation mapping can be used to relate different data types. Chapter 4 illustrates our approach to security analysis, where the problem of finding a security violation is formulated as a constraint satisfaction problem. Chapter 5 introduces our prototype implementation, Poirot, and Chapter 6 describes case studies that involved applying Poirot to analysis of publicly deployed systems. The thesis concludes with discussions of related work in Chapter 7 and future directions in Chapter 8.

Chapter 2

Behavioral Modeling

We will begin by introducing a modeling approach based on a well-known process algebra called communicating sequential processes (CSP) [39]. We will then show how standard CSP can be extended to accomodate the notion of *representations*, and build on this notion to construct a mechanism that can be used to combine distinct but possibly overlapping models of a system.

Chapter Highlights

- A system is modeled as a set of *processes* that interact with each other by engaging in various types of *events* (Section 2.1).
- Each event is assigned one or more labels, each corresponding to a possible *representation* of the event (2.2).
- The overall behavior of a process is defined by a set of *guard* conditions, which describes when a process is allowed to engage in a particular event (2.3).
- A pair of processes can be composed to form a larger process that captures the interaction between the two. During the composition step, a *representation mapping* can be used to introduce a relationship between events that would otherwise be considered completely independent (2.4).
- Establishing such a relationship may introduce new, unanticipated behavior into the system, possibly leading to a security violation (2.5).

2.1 Modeling Systems with Events

Communicating sequential processes (CSP) is a simple but powerful language for describing a system as a set of interacting, concurrent processes [39]. Compared to other popular modeling notations such as state machines, its emphasis on process interaction makes CSP (and process algebras in general) particularly suitable for analyzing *end-to-end* system behavior; that is, how a system behaves as a sum of its parts. As we will see in the next chapter, CSP can be augmented with a simple extension to allow reasoning about flow of information throughout different parts of the system—crucial for any security analysis.

In this approach, a system is modeled as a set of *processes* that interact with each other by engaging in various types of *events*. As a running example throughout this chapter, consider the task of designing a simple online store that provides two basic services to its customers: logging onto the store and adding an item to a shopping cart[1]. Customers interact with the store by sending a series of requests to it, which may decide to serve some of those requests while rejecting others based on their content. For example, when the store notices that a particular item is out of stock, it may decide to reject all future requests for adding the item until it becomes available again.

Events and traces A *process* is an entity that is capable of engaging in certain types of *events*, each of which corresponds to an atomic action in the world. For example, the store may be modeled as a process that participates in two types of events:

- $\texttt{login}(u,p)$: A login of user u using password credential p;
- $\texttt{addItem}(u,i)$: An insertion of item i into the shopping cart owned by user u.

where u, p, and i are parameters of events.

One way to capture the behavior of a process is by treating the process as a black box, and observing events that it performs over a certain period of time. We will assume that no two events can be performed simultaneously by a process. In other words, the set of events observed from the process can be organized into a totally ordered sequence, which we call a *trace* of the process.

For example, one possible trace of the `Store` process consists of the following three events, ordered from left to right:

$$\langle \texttt{login(aliceID, 1234)}, \texttt{addItem(aliceID, choc)}, \texttt{addItem(aliceID, oat)} \rangle$$

where `aliceID` is an identifier for a particular user (named Alice), `1234` is a password value, and `choc` and `oat` are identifiers for two store items, chocolate bars and oatmeal boxes. This trace describes a behavior of the store that begins with logging Alice onto the store, and then adding a chocolate bar followed by an oatmeal box to her shopping cart.

The set of all traces permitted by `Store` captures the *overall* behavior of the store, and is denoted *traces*:

$$\textit{traces}(\texttt{Store}) \in \mathbb{P}(T)$$

where T is the set of all finite sequences of events. Since `Store` may continue to accept requests from customers without ever terminating, *traces*(`Store`) may be potentially infinite.

Composition Typically, the designer would be interested in understanding not just how the store behaves on its own, but also how it interacts with with one or more customers. To explore such composite behaviors, let us first introduce a new process, called `Alice`, which represents the customer with username `aliceID`. Like `Store`, `Alice` is capable of engaging in `login` and `add` events, except Alice does not like oatmeal, and so she would

[1]Of course, a realistic store typically provides many more services, but for simplicity, we will keep our discussion to these two.

never initiate a request for adding a oatmeal box to her shopping cart. In other words, the following trace, consisting of two events, is permitted by `Alice`:

$$\langle \texttt{login(aliceID, 1234)}, \texttt{addItem(aliceID, choc)} \rangle \in \textit{traces}(\texttt{Alice})$$

but the following trace is not a valid behavior of `Alice`:

$$\langle \texttt{login(aliceID, 1234)}, \texttt{addItem(aliceID, oat)} \rangle \notin \textit{traces}(\texttt{Alice})$$

because it includes an `addItem` event that `Alice` would never engage in.

The composition of `Store` and `Alice` itself is a process, and denoted by

$$\texttt{Store} \parallel \texttt{Alice}$$

This process describes a system that consists of the store and Alice executing in parallel with each other. The two interact by *synchronizing* on events that they both are capable of performing. For example, if `Alice` decides to log onto the store by performing `login(aliceID, 1234)`, then `Store` must be willing to accept and perform the same event at that time—otherwise, this event cannot take place in the overall system.

Consider the following trace, which depicts `Alice` logging onto the store and adding `choc` twice to her shopping cart:

$$\langle \texttt{login(aliceID, 1234)}, \texttt{addItem(aliceID, choc)}, \texttt{addItem(aliceID, choc)} \rangle \in \textit{traces}(\texttt{Store} \parallel \texttt{Alice})$$

This is a valid trace of (`Store` || `Alice`), since both `Alice` and `Store` are able to synchronize on each event appearing in the trace. However, the following trace, while permitted by `Store` alone, is not allowed in their composition:

$$\langle \texttt{login(aliceID, 1234)}, \texttt{addItem(aliceID, choc)}, \texttt{addItem(aliceID, oat)} \rangle \notin \textit{traces}(\texttt{Store} \parallel \texttt{Alice})$$

because `Alice` would never agree to participate in `addItem(aliceID, oat)`.

Everything discussed so far—the notions of processes, events, and synchronization—is part of standard CSP [39]. In the next section, we will show how these notions can be extended to facilitate modeling of different views of a system, and to enable a kind of composition that would not be readily expressible in CSP.

2.2 Representations

Suppose that the designer is satisfied with the high-level design of the store, and wishes to move onto the next step of the development: determining how the store services will be implemented. In particular, the designer decides to deploy the store as a standard HTTP server, with customers interacting with the store through a web browser.

In the final deployed system, the store will take on two **different representations, depending on the perspective of the client that interacts with it**. As far as customers are concerned, it will appear to be a typical online shopping cart that provides the basic store-related services. However, to a browser, the store behaves like a web server that accepts HTTP requests at certain designated URLs. The customers are oblivious to the underlying details of the HTTP protocol; on the other hand, particulars of a store item are treated by the browser as nothing more than HTTP packets to be transmitted over the web.

Our goal is to allow a particular aspect of a system to be described in independent models (possibly by different stakeholders), and those models to be brought together in a way that preserves both characterizations of the system. In this section, we will show how standard CSP can be extended to achieve this goal by (1) allowing each event to be associated with multiple representations within a single model, and (2) relating distinct representations from two different models through a *representation mapping*.

2.2.1 Describing Events with Representations

A *representation* of an event, $r \in R$, is simply one possible description of the event, specifying its name and a list of parameters.

The key idea behind our approach is to allow every event to be associated with multiple representations; more precisely, every event is specified as a set of representations; i.e.,

$$E = \mathbb{P}(R)$$

As a consequence, a trace is now a sequence of sets of representations.

The simplest case is when every event contains exactly one label. Recall the following trace from Section 2.1, allowed by the `Store` process:

$$\langle \texttt{login(aliceID, 1234)}, \texttt{addItem(aliceID, choc)}, \texttt{addItem(aliceID, oat)} \rangle$$

With our new definition of events, this trace is now written as

$$\langle \{\texttt{login(aliceID, 1234)}\}, \{\texttt{addItem(aliceID, choc)}\}, \{\texttt{addItem(aliceID, oat)}\} \rangle \in \mathit{traces}(\texttt{Store})$$

More complex cases arise when two or more representations, originating from independent models of a system, are associated with the same event in the world. Let `Server` be a process that depicts the behavior of a standard HTTP server, engaging in exactly one kind of events—HTTP requests. To describe these events, we will introduce a set of labels in the form of

$$\texttt{req}(u, h, b)$$

where u, h, and b are parameters for the URL, headers, and body of an HTTP request, respectively.

The `Server` process describes a *generic* HTTP server, meaning, by itself, it is ready to engage in any arbitrary HTTP request. But as the designer of the store, we would be interested in constructing a more specialized server that accepts two specific kinds of HTTP requests: `login` and `addItem`. To achieve this, we may allocate a certain set of URLs for these specialized HTTP requests. For example, the following URL may be designated for the operation of adding choc to Alice's shopping cart:

$$\texttt{url}_{\texttt{aliceChoc}} = \texttt{http://www.mystore.com//addItem?user=alice\&item=choc}$$

Then, HTTP requests for this operation can be represented as

$$\texttt{req}_{\texttt{aliceChoc}} = \texttt{req}(\texttt{url}_{\texttt{aliceChoc}}, \_, \_)$$

where the headers and body are irrelevant to the request (and thus, denoted by the "don't care" variable, _).

Recall the original Store process, which is willing to engage in events that are represented as follows:

$$\texttt{add}_{\texttt{aliceChoc}} = \texttt{addItem}(\texttt{aliceID}, \texttt{choc})$$

Suppose that the designer wishes to construct a new process (called StoreServer) that corresponds to the deployment of Store as an HTTP server. Ideally, we want to allow the designer to be able to (1) leverage the domain knowledge already captured by the Server process, instead of directly modifying Store and (2) perform the construction of StoreServer in a modular fashion, so that the interaction of Store with its environment (i.e., Alice) needs not be modified.

Every event performed by StoreServer can be associated two different representations: one being the abstract representation from the Store process, and the other one being its concrete realization as an HTTP request. For example, for any event e that results in choc being added to Alice's cart, the following holds true:

$$e = \{\texttt{add}_{\texttt{aliceChoc}}, \texttt{req}_{\texttt{aliceChoc}}\}$$

As we will see later in this chapter, this multi-faceted characteristic of events allows different types of processes to interact with StoreServer without being aware of each other. Customers can continue to make use of the store services without knowing that they are implemented as HTTP requests; similarly, a browser may communicate to StoreServer without being aware of the application-level semantics that the server implements.

In the rest of this chapter, we will discuss (1) how the relationship between a pair of processes (e.g., Store and Server) may be specified, (2) and given this relationship, how our composition mechanism constructs their composition (StoreServer) that maintains the characteristics of the original pair as multiple representations.

2.3 Behavior and Representations

Before introducing our composition mechanism, let us first discuss how the behavior of a process may be influenced by the assignment of potentially multiple representations to its events.

Recall that the overall behavior of process p is defined as a (potentially infinite) set of traces that are permitted by p:

$$\mathit{traces}(p) \subseteq T$$

Traces can be constructed inductively, by taking an existing trace and appending an event that p may choose to perform at that point. For example, consider the following trace from `Alice`:

$$\langle \{\texttt{login(aliceID, 1234)}\}, \{\texttt{addItem(aliceID, choc)}\} \rangle \in \mathit{traces}(\texttt{Alice})$$

Having performed these events, Alice may choose to purchase another chocolate, thus resulting in a new trace:

$$\langle \{\texttt{login(aliceID, 1234)}\}, \{\texttt{addItem(aliceID, choc)}\}, \{\texttt{addItem(aliceID, choc)}\} \rangle$$
$$\in \mathit{traces}(\texttt{Alice})$$

Whether or not a particular event can be appended to an existing trace depends on (1) the characteristics of that event as described by its representations, and (2) the events that already exist in the trace. For example, let us say that our store implements a rather draconian measure, where it restricts each customer from buying more than two copies of an item; when it receives a request from a customer for adding item i, it looks at the list of `addItem` events that it has already performed, and only accepts the new request if i has not been added to the customer's cart twice.

One way to specify this aspect of the process behavior is to express it as a *guard* condition over an existing trace (t) and the event to be added (e). For example, the `Store` process may use the following condition to test whether or not an incoming request should be considered a valid `addItem` event:

$$\mathit{guard}_{\texttt{addItem}}(e, t) \equiv$$
$$\exists\, u \in \texttt{UserID}, i \in \texttt{ItemID} \bullet \texttt{addItem}(u, i) \in e \land \texttt{numAdded}(u, i, t) < 2$$

where $\texttt{numAdded}(u, i, t)$ is an auxiliary function that computes the number of occurrences of $\texttt{addItem}(u, i)$ in trace t. In other words, event e is considered a valid `addItem` and may be appended to trace t if and only if the item being requested has not bee added to the customer u's shopping cart more than once during the execution of t.

Similarly, `Store` may use the following guard condition to define what it considers to be valid `login` requests:

$$\mathit{guard}_{\texttt{login}}(e, t) \equiv$$
$$\exists\, u \in \texttt{UserID}, p \in \texttt{Password} \bullet \texttt{login}(u, p) \in e \land (u, p) \in \texttt{passwords}(t)$$

where `password` is an auxiliary function that computes the set of username-password pairs that are maintained by the store after trace t. Informally, this guard says that e is a valid Login request if and only if the provided p is the correct password for user u.

Given these two guards, we can now specify exactly when an event may be added to a trace of Store:

$$\begin{aligned}\forall t \in T, e \in E \bullet t ^\frown \langle e \rangle \in \mathit{traces}(\texttt{Store}) \Leftrightarrow \\ t \in \mathit{traces}(\texttt{Store}) \wedge \\ \forall r \in e \bullet r \in \texttt{AddItem} \Rightarrow \mathit{guard}_{\texttt{addItem}}(e,t) \wedge \\ r \in \texttt{Login} \Rightarrow \mathit{guard}_{\texttt{login}}(e,t)\end{aligned}$$

where $t_1 ^\frown t_2$ is the concatenation of traces t_1 and t_2, and AddItem and Login are the sets of all addItem and login representations, respectively. In other words, event e may be appended to an existing trace t if and only if e satisfies the guard that is associated with its representation type.

Given the above specification, we may conclude that the following is not a valid trace of Store:

$$\langle \{\texttt{addItem(aliceID,choc)}\}, \{\texttt{addItem(aliceID,choc)}\}, \{\texttt{addItem(aliceID,choc)}\} \rangle \\ \notin \mathit{traces}(\texttt{Store})$$

since the last event does not satisfy $\mathit{guard}_{\texttt{addItem}}$.

When an event is associated with multiple representations, it must satisfy *all* of the guards that are imposed on the types of those representations. The intuition behind this requirement is that, when an event takes on multiple roles, it possesses the characteristics of all of those representations—meaning, it will be subjected to all of the treatments that the process imposes on those representations.

For example, the guard imposed by Server on HTTP requests may say that the host section of the given URL must match the IP address of the server receiving the request:

$$\begin{aligned}&\mathit{guard}_{\texttt{req}}(e,t) \equiv \\ &\quad \exists u \in \texttt{URL}, h \in \mathbb{P}(\texttt{Header}), b \in \texttt{Body} \bullet \texttt{req}(u,h,b) \in e \wedge \mathit{match}(\texttt{host}(u), \texttt{IPaddr}(t))\end{aligned}$$

Here, $\texttt{host}(u)$ returns the host section of given url u; $\texttt{IPaddr}(t)$ returns the IP address associated with Store after trace t, and; match(x,y) returns true if and only if hostname x resolves to the IP address y.

Recall, from the previous section, the event performed by StoreServer that leads to the insertion of choc into Alice's cart; this event is labeled with two different representations:

$$e = \{\texttt{add}_{\texttt{aliceChoc}}, \texttt{req}_{\texttt{aliceChoc}}\}$$

For this event to appear in a trace of StoreServer, it must not only be a valid addItem event, but its encoding as an HTTP request must include the hostname that resolves to the IP address of StoreServer. In other words, e must satisfy the conjunction of the two guards

$$\mathit{guard}_{\texttt{addItem}}(e,t) \wedge \mathit{guard}_{\texttt{req}}(e,t)$$

in order to appear in a trace of StoreServer after it has executed the events in t.

More generally, the overall behavior of process p, denoted by $\mathit{traces}(p)$, can be defined inductively as follows:

$$\begin{aligned} traces(p) = \{t \in Trace \mid\ & t = \langle\rangle \vee \\ & \exists\, t' \in traces(p), e \in E \bullet \\ & \quad t = t' \frown \langle e \rangle \wedge \\ & \quad \forall\, r \in e \bullet guard_{typ(r)}(e, t)\} \end{aligned}$$

where $\langle\rangle$ is an empty trace, $typ(r)$ returns the type of representations that r belongs to, and $guard_t$ is the guard imposed by p on the representation type t. In other words, a new trace t can be constructed by taking an existing trace $t' \in traces(p)$ and appending a new event e that satisfies all of the guards that are imposed on its representations.

2.4 Composition

2.4.1 Parallel Composition

At a high level, our composition approach follows the same basic rule as the parallel composition in standard CSP: A pair of processes, when put together, must simultaneously perform the classes of events that are common to both of them.

Consider two processes, p and q, where p is capable of performing three kinds of events, each labeled with representation a, b, or c. The following is a possible trace of p:

$$\langle \{\mathtt{a}\}, \{\mathtt{c}\}, \{\mathtt{b}\} \rangle \in traces(p)$$

Similarly, q is capable of performing two kinds of events, labeled with c or d; it may generate a trace like this one:

$$\langle \{\mathtt{d}\}, \{\mathtt{c}\} \rangle \in traces(q)$$

Since events labeled with c are common to p and q, each $\{\mathtt{c}\}$ requires simultaneous participation from both processes. For example, given the above two traces, we may conclude that the following is a valid trace of $p \parallel q$:

$$\langle \{\mathtt{a}\}, \{\mathtt{d}\}, \{\mathtt{c}\}, \{\mathtt{b}\} \rangle \in traces(p \parallel q)$$

Conceptually, this trace describes a system execution where the events take place in the following order:

1. p performs $\{\mathtt{a}\}$,
2. q performs $\{\mathtt{d}\}$,
3. p and q simultaneously perform $\{\mathtt{c}\}$, and
4. p performs $\{\mathtt{b}\}$.

The other kinds of events beside $\{\mathtt{c}\}$ are unique to each process, and have no influence on each other; they may be interleaved freely by $(p \parallel q)$. For instance, a different trace, where

$\{\mathtt{d}\}$ is performed before $\{\mathtt{a}\}$ instead, is also a valid behavior of $p \parallel q$:

$$\langle \{\mathtt{d}\}, \{\mathtt{a}\}, \{\mathtt{c}\}, \{\mathtt{b}\} \rangle \in \mathit{traces}(p \parallel q)$$

Our approach takes a departure from standard CSP, in that each event may be labeled with multiple representations. Consider a variant of p, called p', where some events are labeled with two representations, c and x;

$$\langle \{\mathtt{a}\}, \{\mathtt{c}, \mathtt{x}\}, \{\mathtt{b}\} \rangle \in \mathit{traces}(p')$$

Similarly, let q' be a process that assigns two representations, c and y, to some of its events:

$$\langle \{\mathtt{d}\}, \{\mathtt{c}, \mathtt{y}\} \rangle \in \mathit{traces}(q')$$

When two events from distinct processes share *at least one* label, those events can be treated as the same kind of event, and require simultaneous participation from both processes. Intuitively, $\{\mathtt{c}, \mathtt{x}\}$ may be treated like c or x, depending on the perspective of an external process that wishes to interact with p'. Since q' is capable of engaging in $\{\mathtt{c}, \mathtt{y}\}$, which itself can be treated like c, p' and q' possess an ability to influence each other through simultaneous participation in $\{\mathtt{c}, \mathtt{x}\}$ and $\{\mathtt{c}, \mathtt{y}\}$.

When processes synchronize on a pair of events with distinct but overlapping sets of labels, the pair are combined into a new event by computing the *union* of the two label sets. For instance, when p' and q' interact by performing $\{\mathtt{c}, \mathtt{x}\}$ and $\{\mathtt{c}, \mathtt{y}\}$, the shared event takes on the following form in $p' \parallel q'$:

$$\{\mathtt{c}, \mathtt{x}, \mathtt{y}\}$$

So, the following is a valid trace of $p' \parallel q'$:

$$\langle \{\mathtt{a}\}, \{\mathtt{d}\}, \{\mathtt{c}, \mathtt{x}, \mathtt{y}\}, \{\mathtt{b}\} \rangle \in \mathit{traces}(p' \parallel q')$$

while the following trace is not a valid behavior of $p' \parallel q'$:

$$\langle \{\mathtt{a}\}, \{\mathtt{d}\}, \{\mathtt{c}, \mathtt{x}\}, \{\mathtt{c}, \mathtt{y}\}, \{\mathtt{b}\} \rangle \notin \mathit{traces}(p' \parallel q')$$

because $\{\mathtt{c}, \mathtt{x}\}$ cannot occur alone in $p' \parallel q'$ without being synchronized with $\{\mathtt{c}, \mathtt{y}\}$.

2.4.2 Representation Mapping

Our discussion so far has focused on one particular kind of composition, where two processes synchronize on the groups of events that share some common characteristics. Sometimes, a pair of process models may describe the same aspect of a system in reality, but are specified using two completely distinct sets of vocabulary terms. This type of mismatch has been frequently observed in software development, where different (but overlapping) aspects of a system are documented by independent stakeholders, often in isolation from each other. Eventually, to enable an end-to-end analysis, these artifacts need to be integrated into a single coherent model of the system.

The problem is that these models are not readily amenable to the above composition technique; since their events do not share common representations, we would end up with a system in which the processes behave completely independent of each other. A different composition mechanism is needed, where a relationship between a pair of processes, indicating the commonalities that they share in reality, can be specified by the designer before composition takes place.

Let us propose a new composition operator

$$p \underset{m}{\|} q$$

which introduces a relationship between distinct groups of events as specified by the *representation mapping* m, and allows p and q to interact through those events. A *representation mapping* is a relation of type $R \times R$, where $(a, b) \in m$ means that "every a event should also be considered a b". More precisely, the mapping indicates that every event labeled with a must be assigned b as an additional representation in the composite process, $p \| q$.

Recall the sample traces permitted by the processes p and q from the previous section:

$$\langle\{\mathtt{a}\},\{\mathtt{c}\},\{\mathtt{b}\}\rangle \in \mathit{traces}(p)$$
$$\langle\{\mathtt{d}\},\{\mathtt{c}\}\rangle \in \mathit{traces}(q)$$

As we noted, in standard parallel composition, the two processes synchronize on $\{\mathtt{c}\}$ while independently performing events that are unique to themselves, allowing a trace like the following in the resulting composition:

$$\langle\{\mathtt{a}\},\{\mathtt{d}\},\{\mathtt{c}\},\{\mathtt{b}\}\rangle \in \mathit{traces}(p \| q)$$

Suppose that we wish to express a relationship between a pair of distinct representations, a and d, where d is a valid alternative representation for a. This can be specified as an entry in the representation mapping m to be used during the composition of p and q:

$$(a, d) \in m$$

The mapping signifies that every a should be treated like d as well. Consequently, whenever p performs $\{\mathtt{a}\}$, q is required to synchronize with p by simultaneously performing $\{\mathtt{d}\}$. So, the following is a valid trace of $p \underset{m}{\|} q$:

$$\langle\{\mathtt{a},\mathtt{d}\},\{\mathtt{c}\},\{\mathtt{b}\}\rangle \in \mathit{traces}(p \underset{m}{\|} q)$$

whereas the following is not:

$$\langle\{\mathtt{a},\mathtt{d}\},\{\mathtt{c}\},\{\mathtt{b}\},\{\mathtt{a}\}\rangle \notin \mathit{traces}(p \underset{m}{\|} q)$$

since $\{\mathtt{a}\}$ cannot occur alone without being synchronized with $\{\mathtt{d}\}$.

More generally, the set of traces allowed by the composition of processes p and q using mapping m can be defined as shown in Figure 2-1. The intuition behind this definition is as follows. Since every event in t must have been performed by either p or q (or both), erasing events from t that do not share any representations with q must yield a trace of p (a similar argument holds for q as well). In addition, any event with a representation that

Let $t \upharpoonright A$ be a *projection* of trace t onto a set of representations, A; this operator is defined as follows:

$$\begin{aligned}
\langle\rangle \upharpoonright A &= \langle\rangle \\
(t ^\frown \langle e\rangle) \upharpoonright A &= (t \upharpoonright A) ^\frown \langle(e \cap A)\rangle && \text{if } (e \cap A) \neq \emptyset \\
(t ^\frown \langle e\rangle) \upharpoonright A &= (t \upharpoonright A) && \text{otherwise}
\end{aligned}$$

Then, the set of traces allowed by $p \underset{m}{||} q$ can be defined as:

$$\begin{aligned}
traces(p \underset{m}{||} q) = \{t \in T \mid \forall e \in events(t) \bullet e.m \subseteq e \wedge \\
(t \upharpoonright \alpha(p)) \in traces(p) \wedge \\
(t \upharpoonright \alpha(q)) \in traces(q)\}
\end{aligned}$$

where $events(t)$ is the set of events appearing in trace t, and $\alpha(p)$ is the set of all representations belonging to the events of p.

Figure 2-1: Traces allowed by the composition of p and q with representation mapping m.

is mapped to another representation in the mapping m must, by construction, contain the latter as one of its labels.

It is important to point out that the representation mapping is not symmetric; $(a, b) \in m$ does **not** imply that every event labeled b is necessarily treated as a. In our example, while q is required to provide {d} for every {a} performed by p, q is still free to perform {d} on its own, independent of p. Suppose that the following is another trace allowed by q:

$$\langle\{\mathtt{d}\}, \{\mathtt{c}\}, \{\mathtt{d}\}\rangle \in traces(q)$$

Then, the following trace is a valid behavior of the composed system:

$$\langle\{\mathtt{a}, \mathtt{d}\}, \{\mathtt{c}\}, \{\mathtt{b}\}, \{\mathtt{d}\}\rangle \in traces(p \underset{m}{||} q)$$

Note that when m is empty, our composition operator produces the same process as the standard parallel composition:

$$p \underset{\{\}}{||} q = p \,||\, q$$

Notation As a shorthand, we will write

$$a \underset{m}{\longmapsto} b$$

to mean that a is mapped to b in the representation mapping m; i.e.,

$$(a, b) \in m$$

When it is clear from the context which mapping we are referring to, we will simply write

$$a \longmapsto b$$

If the relationship between the two representations is symmetric, we will denote it as

$$a \leftrightarrow b \Leftrightarrow a \longmapsto b \wedge b \longmapsto a$$

Laws The composition operator is **commutative**; that is,

$$p \underset{m}{\|} q = q \underset{m}{\|} p$$

Informally, this property can be inferred from the fact that the trace set definition in Figure 2-1 does not depend on which process the events in the mapping originate from.

However, the composition operator, in general, is **not associative**. Let us consider an example with three processes, p, q, and r, which are capable of performing events $\{a\}$, $\{b\}$, and $\{c\}$, respectively. Suppose that we wish to compose these processes in order using two distinct mappings, m_1, and m_2, such that

$$a \underset{m_1}{\longmapsto} b \qquad\qquad a \underset{m_2}{\longmapsto} c$$

Then, it does not necessarily follow that

$$(p \underset{m_1}{\|} q) \underset{m_2}{\|} r = p \underset{m_1}{\|} (q \underset{m_2}{\|} r)$$

because once q and r has been composed, $\{b\}$ and $\{c\}$ are interleaved into separate events in the resulting process; consequently, it will not be possible for a to be synchronized simultaneously with events that contain both b and c as representations.

In the remainder of this section, we will show how this new operator can be used as a basis for modeling different types of system interaction: (1) communication, where the binding of multiple clients to a service is expressed as a mapping between representations, and (2) implementation, where the relationship between an abstract entity and its concrete counterpart is established by linking representations of events across multiple levels of abstraction.

2.4.3 Modeling Communication

For our modeling tasks, we are interested in capturing a class of system interaction where a component provides a set of services to one or more other client components. In this section, we will show how our composition operator $\underset{m}{\|}$ can be used to express this type of *horizontal* composition between different parts of a system.

The simplest case involves a system in which each service provider interacts with exactly one client. We have already seen examples of this, where `Store` provides `login` and `addItem` services to `Alice`. Describing their composition is straightforward; both `Store` and `Alice` may engage in events labeled with `login` or `addItem`, and when brought together, they are required to synchronize on every one of these events.

But typically, there are multiple clients that interact with a single service provider. Let us introduce another customer, named Eve, who also interacts with Store by invoking the login and addItem services. It turns out that she is a rather mischievous character, and will attempt to cause trouble to other customers by getting the store to insert unwanted items into their shopping carts. For example, Eve may attempt to clutter Alice's cart with an additional choc, by getting the store to perform addItem(aliceID, choc) without Alice's initiation.

Let us assume that Alice and Eve do not directly communicate, and so they have no influence on each other's interaction with the store; this means, for example, that Eve could not force Alice to send a request to the store against the latter's will. As far as the store is concerned, all it sees is a sequence of incoming requests, with this sequence being an arbitrary interleaving of requests from Alice and Eve.

One might be tempted to model Eve the same we did with Alice, describing it as a process that engages in two kinds of events, each labeled login or addItem. For example, the following trace could describe a behavior of Eve where she first logs onto the store, adds an item to her own shopping cart, and then attempts to insert the same item into Alice's cart:

$$\langle\{\texttt{login(eveID, 5678)}\}, \{\texttt{addItem(eveID, choc)}\}, \{\texttt{addItem(aliceID, choc)}\}\rangle \in \mathit{traces}(\texttt{Eve})$$

Eve can be composed with Store in the same manner as Alice is; every login or addItem event appearing in a trace of Store || Eve is the result of the simultaneous engagement of the event by both processes.

This formulation becomes problematic when we attempt to construct a system that consists of Store, Eve, *and* Alice. Since Eve and Alice contain events with common representations (i.e., addItem(aliceID, choc)), their composition, Eve || Alice, would require that the two processes synchronize on every such event. But this deviates from how we intend to model client-provider interactions—instead, we want the client processes to be able to send requests to the provider independently of each other!

One solution is to distinguish events based on the identities of the processes that engage in those events. To achieve this, we will introduce a naming convention where each representation of an event is required to include, as a prefix, the name of the process that engages in the event. More formally, for every event e performed by process p, every representation r assigned to e is given a prefix p; i.e.,

$$r = p.\mathit{descr}$$

where *descr* is a description stating the name and parameters of the event.

For example, to distinguish addItem events generated by Alice from those belonging to Eve, we will specify their representations as:

$$\texttt{Alice.addItem(aliceID, choc)}$$
$$\texttt{Eve.addItem(aliceID, choc)}$$

Now, the two processes contain disjoint sets of event labels, and so when they are brought together, they are free to generate addItem requests independently from each other.

The same naming convention applies to the events performed by Store; for example, the following representation is assigned to an event corresponding to the insertion of choc into Alice's cart:

$$\texttt{Store.addItem(aliceID,choc)}$$

But now we are faced with a different problem, where neither Alice nor Eve is able to communicate to Store, because they do not share any labels for events that they may participate in!

In order to allow clients to interact with a provider, we can leverage the representation mapping and introduce a relationship between the event labels of those processes. More precisely, given a pair of client c and service provider s, ready to engage in events labeled $c.descr$ and $s.descr$, respectively, the following entry is specified in the representation mapping:

$$c.descr \underset{m}{\longmapsto} s.descr$$

Then, the resulting composition, $c \underset{m}{||} s$, behaves like a system where every *descr* event generated by c is simultaneously engaged by s.

Back to our example, to allow Alice's requests to be delivered to and served by Store, we will introduce the following relationship between Alice's and the store events:

$$\texttt{Alice.addItem(aliceID,choc)} \underset{m_{Alice}}{\longmapsto} \texttt{Store.addItem(aliceID,choc)}$$

When the two processes are brought together, the resulting system, $\texttt{Store} \underset{m_{Alice}}{||} \texttt{Alice}$, behaves as desired; every Alice.addItem event appearing in one of its traces is also assigned Store.addItem as a representation, indicating that every request from Alice is served by the store.

Recall that the notion of the representation mapping is not symmetric; that is, every Alice.addItem is bound to Store.addItem, but the converse is not necessarily true. Thus, $\texttt{Store} \underset{m_{Alice}}{||} \texttt{Alice}$ may allow a trace like

$$\langle\{\texttt{Store.addItem}\},\{\texttt{Store.addItem},\texttt{Alice.addItem}\},\{\texttt{Store.addItem}\}\rangle \in traces(\texttt{Store} \underset{m_{Alice}}{||} \texttt{Alice})$$

where the first and third events are not bound to any events from clients. Intuitively, these events can be regarded as "open" endpoints, signifying that Store is willing to accept requests from other clients beside Alice.

Let us now bring Eve into the picture. We will allow it to interact with the store by binding addItem requests from Eve to their corresponding events in Store:

$$\texttt{Eve.addItem(aliceID,choc)} \underset{m_{Eve}}{\longmapsto} \texttt{Store.addItem(aliceID,choc))}$$

Then, Eve can be brought as an additional participant into the existing interaction between

Alice and Store:

$$\texttt{StoreSystem} = (\texttt{Store} \underset{m_{Alice}}{||} \texttt{Alice}) \underset{m_{Eve}}{||} \texttt{Eve}$$

The resulting process, denoted StoreSystem, captures the interaction between the store and its two clients as we originally intended; Alice and Eve independently generate addItem requests to the store, which will then decide whether to serve those requests based on their content. For example, the following is a valid trace of StoreSystem:

$$\langle\{\texttt{Store.addItem},\texttt{Eve.addItem}\},\{\texttt{Store.addItem},\texttt{Alice.addItem}\},\{\texttt{Store.addItem}\}\rangle \in \mathit{traces}(\texttt{StoreSystem})$$

where the first and second events represent addItem requests sent by Eve and Alice, respectively, with both being served by Store. Note that StoreSystem may still permit events that are not bound to any clients (like the last event), allowing us to introduce additional clients into the system if needed.

2.4.4 Modeling Implementation Relationships

In this section, we are interested in a different kind of composition—one that involves a pair of processes that describe a common aspect of a system, but at different levels of abstraction. Again, we will use the representation mapping as a way to express the relationship between the two processes; in this case, the mapping embodies the designer's decisions about how an abstract concept is to be implemented as a more concrete entity.

A key insight is that the implementation step can be expressed as a synchronization requirement between a pair of abstract and concrete processes, p_a and p_c. In particular, some subset of the events performed by p_a may be assigned their concrete counterparts in p_b, so that when the two processes are brought together, they are required to synchronize on every one of those events.

Suppose that the designer is satisfied with the high-level design of the store, and wishes to move onto the next step of development: deploying the system on an HTTP client-server architecture. The goal is to construct a specialized version of an HTTP server (let us call it StoreServer) that provides login and addItem services as HTTP requests.

Server is a process that depicts a *generic* HTTP server, engaging in events represented as $\texttt{req}(u, h, b)$, where u, h, and b are parameters for the URL, headers, and body of an HTTP request, respectively. Some of these HTTP requests will be used to implement the abstract store operations. For example, recall that an abstract Store event leading to the insertion of choc into Alice's cart is characterized by the following representation[2]:

$$\texttt{add}_{\texttt{aliceChoc}} = \texttt{addItem}(\texttt{aliceID}, \texttt{choc})$$

This abstract event may be encoded as an HTTP request in a number of different ways. In one possible encoding, we may designate the a particular URL to be the target address for

[2]For simplicity, in this section, we will omit the process-name prefixes normally included in event representations; they are orthogonal to our discussion here.

the `addItem` operation, and transmit the username and the item ID as query parameters:

$$\texttt{url}_{\texttt{aliceChoc}} = \texttt{http://www.mystore.com//addItem?user=alice\&item=choc}$$

The headers and body are irrelevant to this particular encoding, and so we will simply leave them unspecified. Then, following this encoding, an HTTP request that adds `choc` into Alice's cart is characterized as:

$$\texttt{req}_{\texttt{aliceChoc}} = \texttt{req}(\texttt{url}_{\texttt{aliceChoc}}, \_, \_)$$

Having determined how an abstract event is encoded as a more concrete one, the next step is to ensure that the resulting process, `StoreServer`, actually performs the appropriate HTTP event when it receives an abstract request from `Alice` or `Eve`. In other words, every $\texttt{add}_{\texttt{aliceChoc}}$ event performed by `StoreServer` must be treated like $\texttt{req}_{\texttt{aliceChoc}}$ as well. To do this, we will express the relationship between the two representations as an entry in the mapping used during the composition of `Store` and `Sever`:

$$\texttt{add}_{\texttt{aliceChoc}} \underset{m_{Deploy}}{\longmapsto} \texttt{req}_{\texttt{aliceChoc}}$$

This mapping will ensure that every $\texttt{add}_{\texttt{aliceChoc}}$ event is also assigned $\texttt{req}_{\texttt{aliceChoc}}$ as an additional representation in the composite process.

But this does not accurately capture our intended relationship between the abstract and concrete `addItem` operations. Once $\texttt{req}_{\texttt{aliceChoc}}$ is used to implement $\texttt{add}_{\texttt{aliceChoc}}$, the two representations should be bound to each other; that is, whenever `StoreServer` engages in $\texttt{req}_{\texttt{aliceChoc}}$, it may be observed as performing $\texttt{add}_{\texttt{aliceChoc}}$, from the perspective of `Alice` or `Eve`. In other words, the relationship between the two representations is bi-directional, and should be expressed as such in m_{Deploy}:

$$\texttt{add}_{\texttt{aliceChoc}} \longleftrightarrow \texttt{req}_{\texttt{aliceChoc}}$$

With this mapping, we may now construct a process that behaves like the deployment of the store as a HTTP server:

$$\texttt{StoreServer} = \texttt{Store} \underset{m_{Deploy}}{\|} \texttt{Server}$$

In the resulting process, the set of events labeled $\texttt{add}_{\texttt{aliceChoc}}$ is exactly the same as the set of events labeled $\texttt{req}_{\texttt{aliceChoc}}$. Every event in this set possesses the characteristics of both representations, and is able to take on different roles depending on the process that communicates to `StoreServer`. `Alice` and `Eve` will continue to generate an `addItem` request without being aware of its underlying HTTP-related details, whereas a browser will treat it as a HTTP request being served at a designated URL (without necessarily knowing that it implements a particular piece of store functionality).

The `login` operation is also to be implemented as HTTP requests, and so the relationship between the abstract and concrete operations are specified as:

$$\texttt{login}_{\texttt{alice}} \longleftrightarrow \texttt{req}_{\texttt{aliceLogin}} \land \texttt{login}_{\texttt{eve}} \longleftrightarrow \texttt{req}_{\texttt{eveLogin}}$$

where $\texttt{login}_{\texttt{alice}}$ and $\texttt{login}_{\texttt{eve}}$ represent the abstract login events for Alice and Eve, respectively, and $\texttt{req}_{\texttt{aliceLogin}}$ and $\texttt{req}_{\texttt{eveLogin}}$ represent their encodings as HTTP requests.

In this section, we described only a few particular instances of addItem and login events (involving Alice and Eve) and their encodings. In general, the number of possible event parameter combinations is infinite, and so specifying m by explicitly listing all its entries is not a viable task. Later in this thesis (Chapter 4, Section 4.2), we will show that (1) the representation mapping can be specified more concisely by using declarative constraints and (2) some parts of the mapping may be left unspecified, with an analysis used to synthesize a complete mapping that exposes potential vulnerabilities in the system.

2.5 Implications of Multiple Event Representations

We have proposed the representation mapping as a mechanism for enabling interaction between processes that would have otherwise remained completely independent from each other. But sometimes, this type of composition can introduce behaviors that might not have been intended by the designer. Some of these interaction may even be undesirable, allowing a malicious actor to exploit an unanticipated behavior and undermine the security of the system.

Back to our store example, let us introduce a new process, called Browser, which depicts the behavior of a standard HTTP browser. This process attempts to communicate to Server by participating in a series of events that are assigned labels in the form of Browser.req:

$$\langle\{\texttt{Browser.req}_a\},\{\texttt{Browser.req}_b\},\{\texttt{Browser.req}_c\},\ldots\rangle \in \mathit{traces}(\texttt{Browser})$$

To allow Browser to communicate to Server, we will specify the relationship between the two processes in the mapping m_{HTTP}, ensuring that each browser request is properly served:

$$\texttt{Browser.req}_a \mapsto \texttt{Server.req}_a \wedge \texttt{Browser.req}_b \mapsto \texttt{Server.req}_b \wedge \ldots$$

Their composition is a process that captures the generic interaction between an HTTP server and a browser:

$$\texttt{Server} \underset{m_{HTTP}}{\|} \texttt{Browser}$$

Recall, from Section 2.4.3, the communication between Alice and Store, depicted by the following process:

$$\texttt{Alice} \underset{m_{Alice}}{\|} \texttt{Store}$$

The above two composite processes describe system interaction at different levels of abstraction, and are completely isolated from each other; they do not share any event representations, and so are not able to exert any influence over each other's behavior.

But this separation breaks down once the store is deployed as an HTTP server. Let us bring these two systems together, by introducing an implementation relationship between

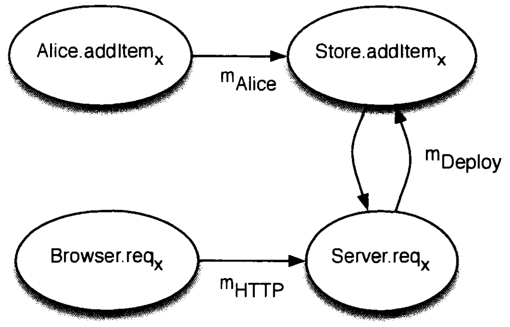


Figure 2-2: A representation graph showing the relationships between event representations in the store example. Each bubble corresponds to an event representation. An edge from r_1 to r_2 means that every r_1 event is also assigned r_2 as an alternative representation; the label on the edge is the representation mapping used to relate the two representations.

Store and Server through m_{Deploy}:

$$(\texttt{Alice} \underset{m_{Alice}}{\|} \texttt{Store}) \underset{m_{Deploy}}{\|} (\texttt{Server} \underset{m_{HTTP}}{\|} \texttt{Browser})$$

which is equivalent to[3]

$$\texttt{Alice} \underset{m_{Alice}}{\|} (\texttt{Store} \underset{m_{Deploy}}{\|} \texttt{Server}) \underset{m_{HTTP}}{\|} \texttt{Browser}$$

which, in turn, may be rewritten as follows:

$$\texttt{DeployedStore} = \texttt{Alice} \underset{m_{Alice}}{\|} \texttt{StoreServer} \underset{m_{HTTP}}{\|} \texttt{Browser}$$

The resulting process describes a system where StoreServer interacts with Alice and Browser through two seemingly separate interfaces. Alice communicates to StoreServer through addItem or login events, whereas Browser communicates by generating HTTP requests. Alice is able to interact with StoreServer without any knowledge of HTTP, and similarly, Browser may not be aware that a particular HTTP request corresponds to some store operation.

However, as a consequence of the composition, Browser is now able to engage in new types of events that were not previously available to it. Figure 2-2 shows a *representation graph,* which depicts the relationships between different representations of events in the system. Suppose that Browser engages in event e with some representation Browser.req$_x$; from the above graph, we may derive the set of representations assigned to e as:

$$e = \{\texttt{Browser.req}_x, \texttt{Store.req}_x, \texttt{Store.addItem}_x\}$$

This means that, whenever Browser sends a request labeled req$_x$, StoreServer will treat it not only like an HTTP request, but addItem$_x$ as well. In other words, Browser is able to in-

[3]As noted in Section 2.4.2, the composition operator is not associative in general; however, in this case, no single representation is mapped to multiple different representations, and thus, it is safe to perform the composition of Store and Server first.

duce StoreServer to perform an addItem operation—an interaction that was not possible prior to the composition.

Imagine that $\texttt{addItem}_x$ corresponds to the insertion of an oatmeal into Alice's shopping cart:

$$\texttt{addItem}_x = \texttt{addItem(aliceID, oat)}$$

Earlier in this chapter when we first introduced Alice, we mentioned that she does not like oatmeal, and so would never initiate a request for adding this item to her cart. Thus, not a single trace of $(\texttt{Alice} \underset{m_{Alice}}{\|} \texttt{Store})$ contains event e' such that

$$e' = \{\texttt{Alice.addItem}_x, \texttt{Store.addItem}_x\}$$

since Alice would refuse to engage in events with label $\texttt{Alice.addItem}_x$. In other words, we may safely assume that the store would never insert an oatmeal box into Alice's cart, as desired by her.

But this assumption no longer holds in DeployedStore, because Browser may initiate a request that causes the store to add a oatmeal box into Alice's cart! For example, the following trace may be permitted by DeployedServer:

$$\langle\{\texttt{Store.login(aliceID, 1234)}\}, \{\texttt{Store.addItem(aliceID, choc)}\}, \{\texttt{Store.addItem(aliceID, oat)}\}\rangle \in \mathit{traces}(\texttt{DeployedServer})$$

where the first two requests are initiated by Alice, whereas the last one is sent from Browser. But as far as the store is concerned, it cannot distinguish where the requests originated from—they will be served equally[4].

While this may seems like a relatively trivial example, it reflects one of the common security mistakes that web developers commit: Making an assumption about clients' behavior on the web. An application may be designed with a dangerous assumption that users will follow its workflow in an intended order, or sends requests only through a narrow front-end interface (e.g., by clicking on hyperlinks or buttons on a page). In fact, however, anyone on the web with an ability to send HTTP requests will be able to directly interact with the application server. For example, in one well-publicized incident, a security researcher was able to access e-mails of any Verizon customer simply by modifying the username field in a URL for its mobile API[5]—suggesting that the developers might have assumed that the users would interact with the app only through its UI.

[4]Note that the process-name prefixes in event representations are used as a modeling idiom to enable composition, not information that is accessible to processes themselves.

[5]http://www.forbes.com/sites/thomasbrewster/2015/01/19/verizon-customer-emails-exposed/

Chapter 3

Dataflow Modeling

Many security properties can be described as a restriction on the flow of certain types of data between different parts of a system. For example, the confidentiality of a customer's shopping cart may be expressed as a constraint that "information on the content of the cart should never be accessible to another customer". To allow the designer to specify such properties easily, we will augment our trace-based model of behavior with a simple notion of dataflow, where data values are passed from one process to another as parameters of an event.

Just as an event can be associated with multiple representations, so can a piece of data that is passed around. In this chapter, we will extend the notion of representations to data values as well, and show how distinct data representations can be related through a representation mapping. We will also discuss security implications of multiple representations, demonstrating how subtle interactions between the design decisions encoded in the mapping may introduce unanticipated vulnerabilities into the system.

Chapter Highlights

- *Data values* are carried from one process to another as parameters of an event (Section 3.1).
- A data value can be assigned multiple representations, each of which provides a possible description of the structure of the data. A representation mapping can be used to relate a pair of distinct data representations during a composition process (Section 3.2).
- A piece of data with multiple representations may be interpreted differently under distinct contexts; certain types of security attacks exploit this discrepancy between different interpretations of data (Section 3.3).

3.1 Events and Data Values

A data value, $v \in V$, is an atomic entity that may be transmitted among different participants of a system. Intuitively, a value can be regarded as flowing from one process to another by **being carried as a parameter of an event.** To make this notion more precise, let us first divide the set of events performed by process p into *input* and *output* events.

$$in(p), out(p) \subseteq E$$

Within a single process, no event can be both an input and output event:

$$in(p) \cap out(p) = \emptyset$$

If a pair of processes p and q engage in event e such that

$$e \in in(p) \qquad e \in out(q)$$

p is called the *sender* of the event, and q is called its *receiver*.

Our notion of dataflow is bi-directional; that is, as a result of an event taking place, a value may follow from the sender to the receiver or vice-versa. More precisely, value v may flow into process p when (1) p receives an input event that carries v as one of its arguments, or (2) p sends an output event that results in v being returned to p. To support this bi-directional notion, we will associate each event e with a set of *argument* and *return* values:

$$args(e), rets(e) \subseteq V$$

As an example, let us introduce another kind of store event, called `DisplayCart`, which takes a username and returns the the content of the cart owned by that user. Let e be a `DisplayCart` event, and $v_{username}, v_{cart}$ be values that correspond to some username and the content of a particular cart:

$$args(e) = \{v_{username}\} \qquad rets(e) = \{v_{cart}\}$$

When `Alice` and `Store` engage in this event, dataflow takes place in two opposite directions: v_{user} is transmitted from `Alice` to `Store`, which then returns v_{cart} back to `Alice`.

Suppose that `Alice` has not sent any `DisplayCart` request prior to engaging in the event e. Assuming that `Store` releases the information about carts through `DisplayCart`, this implies that `Alice` cannot access v_{cart} until she receives it as a return value of e. In other words, the set of information accessible to `Alice` may grow over time as she participates in more events; conversely, she cannot access a piece of information until she performs an event that carries that information.

More generally, building on the above notion of dataflow, we can define what it means for a process to be able to access a piece of data at a certain point in execution. To do this, we will first allow a certain set of values to be accessible to each process at the beginning of the system execution; we say the process *owns* those values:

$$owns(p) \subseteq V$$

For example, `Alice` may be designated as owning a password that she uses for logging onto the store:

$$v_{\texttt{alicePwd}} \in \mathit{owns}(\texttt{Alice})$$

whereas this value may not initially be available to Eve:

$$v_{\texttt{alicePwd}} \notin \mathit{owns}(\texttt{Eve})$$

Then, the function $\mathit{mayAccess}(p,t)$ defines the largest set of values that are accessible to p after trace t:

$$\begin{aligned}\mathit{mayAccess}(p,t) = \{v \in V \mid{} & v \in \mathit{owns}(p) \lor \\ & \exists e \in \mathit{Event} \bullet e \in (t \upharpoonright \mathit{in}(p)) \land v \in \mathit{args}(e) \lor \\ & e \in (t \upharpoonright \mathit{out}(p)) \land v \in \mathit{rets}(e)\}\end{aligned}$$

where $(t \upharpoonright X)$ returns trace t restricted to the set of events in X. In other words, p may gain access to value v if (1) p owns v or (2) p has already received an event that carries v as an argument, or (3) p has sent an event whose return values include v.

Intuitively, before process p can perform an event that carries value v as one of its parameters, p must already have access to v. To be more precise, (1) for every output event that p performs, all of its arguments must be accessible to p, and (2) for each input event, any associated return parameter must be accessible to p before the event can take place. These conditions can be formalized as an axiom and imposed on every process:

$$\begin{aligned}&\forall p \in P, e \in E, t \in \mathit{traces}(p) \bullet \\ &\quad t^\frown \langle e \rangle \in \mathit{beh}(p) \Rightarrow \\ &\qquad (e \in t \cap \mathit{out}(p) \Rightarrow \mathit{args}(e) \subseteq \mathit{mayAccess}(p,t)) \land \\ &\qquad (e \in t \cap \mathit{in}(p) \Rightarrow \mathit{rets}(e) \subseteq \mathit{mayAccess}(p,t))\end{aligned}$$

This axiom rules out spurious flow of data; without it, we would run into rather strange scenarios where a process is able to make up any arbitrary values and transmit them to another process!

The function *mayAccess* can be used as a convenient notion for expressing a wide range of security properties. Consider the following property as an example:

Eve should never be able to access the content of Alice's shopping cart.

Let $\texttt{cart}(u,t)$ be an auxiliary function that returns the current content of a shopping cart (i.e., the set of items) owned by user u after trace t. Then, the above property can be specified as follows:

$$\forall t, t' \in T, c \in V \bullet c = \texttt{cart}(\texttt{aliceID}, t) \land t \leqslant t' \Rightarrow c \notin \mathit{mayAccess}(\texttt{Eve}, t')$$

where $t \leqslant t'$ means t is a prefix of t'. In other words, if c describes the content of Alice's shopping cart after t, Eve should never be able to access that information at any point in future. A security analysis, explained in Chapter 4, then involves finding a counterexample trace that shows how Eve may be able to access the content of Alice's shopping cart.

3.2 Data Representations

In Chapter 2, we introduced the idea of assigning multiple representations to an event, and proposed a mechanism for composing distinct but overlapping views of a system by relating those representations. A similar notion can be applied to data, by separating a data value from its representations, and allowing multiple representations to be assigned to a value.

Intuitively, a single value may be described in multiple different representations, depending on the task at hand or the perspective of the person observing the entity. Consider the notion of *credential*, which is used as an attestation of claims about a user's identity or capabilities within a system. Over the course of a system lifecycle, a credential may be represented as an abstract token without any internal structure (like passwords from our store example); a browser cookie used for client authentication; an IP packet transmitted over a network; a `String` object inside a Java application; a database record; or a sequence of bytes stored on a hard disk.

Typically, representations of an entity are not completely independent of each other, and some relationships exist among them. A browser cookie, when transmitted over a network, does not map to any arbitrary IP packet; instead, a mechanism in the network stack determines how a given piece of data is to be encoded as a particular sequence of packets. Like we have done with events, we will use a representation mapping to specify such relationships between different representations of a value.

Given one particular representation r of value v, its alternative representations can be obtained by navigating through the representation mapping from r. Having access to a browser cookie, we will be able to obtain its corresponding IP packet(s) by examining the encoding mechanism, which is captured by the representation mapping. Conversely, given a sequence of IP packets that encode a particular cookie, we can obtain the latter by navigating backward through the mapping.

What are the implications of multiple representations on security? A representation may contain details about an entity that are absent from its other representation. These additional details, in turn, may be exploited by an attacker to manipulate the system into producing an undesirable behavior. Later in this chapter, we will explore one example of such attack in detail.

3.2.1 Describing Data with Multiple Representations

A more secure store design Before discussing data representations, let us first improve our original design of the store by introducing the notion of *authentication tokens*. In this new design, when a customer logs onto the store, she is given a special token that she will be asked to provide in subsequent interactions with the store. When the store receives an `addItem` or `displayItem` request, it uses the provided token to identify the user associated with the request. More specifically, the new `Store` process may now engage in the following types of events:

- `login`(u,p,t): Logins in user u with password p, returning authentication token t to be used for subsequent requests;

- `addItem`(t,i): Inserts item i into the shopping cart of the user identified by token t;

- `displayItem`(t,c): Returns the content c of the shopping cart belonging to the user identified by token t.

The major difference from the previous design is that the authentication token must be first obtained from a `login` event before being able to perform other events. Presumably, this should prevent Eve from adding an item to Alice's shopping cart, since Eve would not be able to obtain Alice's token without knowing her password.

Data representations A representation of a data value, $r \in R_V$, is simply one possible description of its structure, embodying information about its *type* and a set of *fields*, which defines its internal structure in terms of other data representations.

A *data type* is a set of similar representations grouped together. For example, we may introduce a data type called `Token` $\subseteq R_V$, which contains a set of tokens used by the store to authenticate its customers. Let us assume that each token is assigned a field with a unique numeric ID to distinguish itself from other tokens[1]:

$$\texttt{Token} = \{\texttt{token}(1), \texttt{token}(2), ...\}$$

Similarly, for our HTTP model, we may introduce a data type called `Cookie`, which refers to a set of cookies that are used to pass information from a browser to a web server. Each cookie is structured like as follows:

$$\texttt{cookie}(name, val)$$

where *name* and *val* are `String` fields that correspond to the name and value of this cookie.

Like we did with events, we will allow each data value to be associated with multiple representations:

$$V = \mathbb{P}(R_V)$$

In addition, we will use the concept of a representation mapping, $m_V \subseteq R_V \times R_V$, to introduce a relationship between a pair of distinct data representations [2]. More specifically, $(a, b) \in m_V$ indicates that every data value labeled a can also be treated like b.

Composition with data representations The composition of a pair of processes may now involve relating not only distinct representations of an event, but those of a data value as well. To take this into account, we will modify our composition operator so that it now accepts a pair of representation mappings, m_E and m_V:

$$p \underset{M}{||} q$$

where $M = (m_E, m_V)$. Formally, the set of data values belonging to to the composition of p and q with the mapping m_V can be defined in a similar manner as the trace set of the

[1]For convenience, we will assume the existence of `Integer` and `String` as built-in primitive data types.

[2]To distinguish these concepts from their counterparts for events, we will refer to them as R_E and m_E; however, the subscript will be omitted when it is clear from the context which function we are referring to.

composition (from Figure 2-1):

$$owns_{(p\underset{M}{\|}q)} = \{v \in V \mid v.m_V \subseteq v \wedge (v \cap \alpha_V(p)) \in owns(p) \wedge (v \cap \alpha_V(q)) \in owns(q)\}$$

where $\alpha_V(p)$ is the set of all data representations appearing in the values owned by p. In other words, every data value v in the composition can be decomposed into two values, each belonging to p and q; in addition, if v contains a representation that is mapped to another by m_V, then v, by construction, must contain the latter as one of its labels.

Example Let $v_{\texttt{at}}$ be a value that corresponds to the authentication token allocated for Alice. In the high-level design of the store, this value may be described as an abstract entity that is associated with a unique, randomly chosen alphanumeric ID:

$$v_{\texttt{at}} = \{\texttt{token(abc42)}\}$$

During the deployment of the store onto an HTTP server, one of the key decisions that the designer must make is to determine how the abstract `Token` data will be represented at the HTTP layer. Suppose that her decision is to store and transmit the token as a browser cookie inside HTTP requests. There are multiple ways to encode `Token` as `Cookie`; in one straightforward encoding, each cookie is assigned a fixed name (e.g., "authToken"), and its value is set to the literal translation of the token ID into a string. So, for example, the encoding of Alice's token as a cookie may look like

$$\texttt{cookie(``authToken'',``abc42'')}$$

This decision can be specified as an entry in m_V, stating that every piece of data represented as `token(abc42)` should also be treated like its cookie counterpart:

$$\texttt{token(abc42)} \underset{m_V}{\longmapsto} \texttt{cookie(``authToken'',``abc42'')}$$

But it would be quite tedious to specify m_V by explicitly listing all of its entries; instead, the general relationship between a token and its cookie encoding may be specified as a constraint, such as

$$\begin{aligned} & m_V \cap (\texttt{Token} \times \texttt{Cookie}) = \\ & \quad \{\texttt{token}(id), \texttt{cookie}(name, val) \mid name = \texttt{``authToken''} \wedge val = \texttt{toString}(id)\} \end{aligned}$$

This mapping can then be used as part of the composition step where `Store` is deployed onto `Server`:

$$\texttt{StoreServer} = \texttt{Store} \underset{M}{\|} \texttt{Server}$$

In the resulting process, every value corresponding to Alice's token is assigned two alternative representations; i.e.,

$$v_{\texttt{at}} = \{\texttt{token(abc42)}, \texttt{cookie(``authToken'',``abc42'')}\}$$

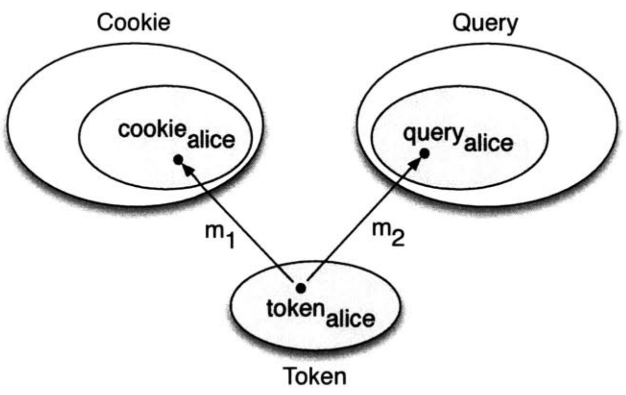


Figure 3-1: Two alternative ways of encoding an authentication token—a browser cookie or a URL query parameter. A circle denotes a set of representations, and an edge denotes an entry in the representation mapping. In the deployed store, every token is also treated like a cookie or a query parameter, depending on the choice of the encoding.

Another example In an alternative design, as shown in Figure 3-1, an authentication token may be embedded as a *query parameter* inside a URL instead of a cookie. Let us look at the structure of an URL:

$$\texttt{url}(origin, path, queries)$$

Each URL contains an origin (itself consisting of a protocol, a hostname, and a port), a path, and a set of query parameters. Every query parameter is a pair of name-value strings:

$$\texttt{query}(name, val)$$

To encode a token as a query parameter, the designer may adapt a simple scheme where the token ID is directly translated into a string and assigned to the value field of a query parameter; so, for example, `token(abc42)` may be embedded into a URL as follows:

$$\texttt{url}(\texttt{origin}_{\texttt{store}}, \texttt{path}_{\texttt{displayCart}}, \{\texttt{query}(``\texttt{authToken}", ``\texttt{abc42}")\})$$

where $\texttt{origin}_{\texttt{store}}$ is a particular origin that identifies the store server (e.g., `http://www.mystore.com`), and $\texttt{path}_{\texttt{displayCart}}$ is a path that points to the HTTP action for displaying a cart (e.g., `/displayCart`). In the real system, this `url` instance would represent

```
http://www.mystore.com/displayCart?authToken=abc42
```

The two alternative encodings (cookie vs. query parameter) appear to be functionally identical, in that they both fulfill the task of transmitting a token inside an HTTP request. However, as we will discuss in Section 3.3, the choice of encoding can have significant impact on the security of the system.

3.2.2 Relating Events, Data, and Representations

Let us revisit the concept of *event representations*, each of which contains the name of an event and a set of parameters. For example, an event that involves inserting a chocolate

$$\begin{tikzcd}
r_e \arrow[r, "args_R{,}rets_R"] & r_V \\
e \arrow[u, "representation"] \arrow[r, "args{,}rets"'] & v \arrow[u, "representation"']
\end{tikzcd}$$

Figure 3-2: Commuting diagram relating events and values to their representations.

into Alice's shopping cart can be represented as:

$$e = \texttt{addItem(aliceToken, choc)}$$

Here, the parameter `aliceToken` represents the authentication token for Alice, and `choc` the item that she wishes to add to her shopping cart.

But how are these parameters related to the actual values that are carried with the event? A simple relationship exists between them: The parameters of an event representation are themselves representations of the values that are associated with the event. Recall that each event e is associated with a set of argument and return values (*args* and *rets*). We will lift these two concepts so that each event representation, r_e, is itself associated with a set of argument and return data representations:

$$args_R(r_e), rets_R(r_e) \subseteq R_V$$

Then, the relationship among events, values, and their representations can be illustrated with a commuting diagram, as shown in Figure 3-2. Starting with some event e, there are two ways of navigating to a particular representation, r_V, that describes a piece of data transmitted along with e: (1) by finding a representation of the event, r_e, that contains r_v as one of its parameters, or (2) by retrieving a parameter value, v, that is labeled with r_v as one of its representations.

3.3 Implications of Multiple Data Representations

Consider a data value, v, labeled with two distinct representations, a and b:

$$v = \{a, b\}$$

The value v possess the characteristics of both a and b, and so it may be interpreted like a or b, depending on the process that gains access to the data. Often, the process will only consider the representation that it deems relevant to its own operations, and may not even be aware that the same data could be interpreted in different ways by other processes. For instance, `Browser` treats all cookies equally, regardless of whether they are used as an authentication token or a simple mechanism for transmitting data between a server and a client. On the other hand, from the perspective of `Store`, that an authentication token is encoded as a cookie is an implementation detail that may be considered inconsequential to the store operation.

Many common security attacks, we argue, exploit this discrepancy between different interpretations of a data value or an event under distinct contexts. Let us look at one example of such attacks in detail.

3.3.1 Cross-Site Request Forgery (CSRF)

Security property One desirable property of the online store is the *integrity of shopping carts*:

> *A shopping cart should only contain items that its owner intends to purchase.*

In other words, a malicious actor should not be able to sabotage another customer's cart by inserting arbitrary items into it. To specify this property formally, we will first introduce an auxiliary function, called `shoplist`, which associates a customer process with the IDs of the items that she wishes to purchase:

$$\texttt{shoplist(Alice)} \subseteq \texttt{ItemID}$$

Earlier, we stated that Alice likes chocolates but not oatmeals, and so

$$\texttt{choc} \in \texttt{shoplist(Alice)} \qquad \texttt{oat} \notin \texttt{shoplist(Alice)}$$

The content of the shopping cart, represented by a data type called `Cart`, consists of a set of item IDs[3]:

$$\texttt{cart}(\mathit{items}) \in \texttt{Cart}$$

Then, the integrity property for Alice's cart can be expressed as

$$\begin{aligned} &\forall t \in T, i \in \texttt{ItemID}, c \in \texttt{Cart} \bullet \\ &\qquad c = \texttt{getCart(aliceID},t) \land i \in c.\mathit{items} \Rightarrow i \in \texttt{shoplist(Alice)} \end{aligned}$$

where $\texttt{getCart}(u,t)$ returns the content of the shopping cart belonging to user u after trace t. Informally, the above formula says that every item in Alice's cart must always be one of the items that she intends to purchase. The attacker's goal is to undermine this property, by getting the store to insert an item into Alice's cart that she does not intend to purchase (e.g., an oatmeal box).

Browser behavior Let us elaborate the structure of an HTTP request to be a little more accurate than we earlier described, introducing two additional parameters—the method type (*method*) and the response to the request (*resp*):

$$\texttt{req}(\mathit{method}, \mathit{url}, \mathit{headers}, \mathit{body}, \mathit{resp})$$

[3]This does not allow duplicate item entries, but for our discussion at hand, we will ignore that aspect.

The response itself consists of three fields—an HTTP status code, a set of response headers and a web resource (e.g., an HTML document, a JSON object, etc.,):

$$\texttt{respMsg}(\mathit{status}, \mathit{headers}, \mathit{resource})$$

Previously, we introduced `Browser` as a process that engages in HTTP request events with `Server`. The actual behavior of a standard browser is more intricate; in particular, the browser places various restrictions on the kinds of HTTP requests that can be generated under different circumstances. For example, not only does it generate a request when a user types a URL into the address bar, it may trigger an additional request when an HTML page is loaded (e.g., to fetch an external image), or when a previous request is redirected to another URL. In our approach to modeling the browser, we will classify these requests into different types of events[4]:

- `userReq`: A request explicitly initiated by the user, either by typing a URL into the address bar, clicking on a hyperlink, or submitting a form.
- `redirectReq`: Generated when a previous request is redirected to another URL.
- `renderingReq`: Triggered when rendering an HTML page with tags that reference external resources, such as `img` or `script` tags.

These requests all share the same set of parameters as `req` does, but each of them is associated with a distinct guard condition that determines when the event may take place. For example, a redirect request is triggered when a previous request returns the HTTP status code 301, with a *location* header that indicates the URL at which the request should be redirected; this condition can be expressed as follows:

$$\begin{aligned}
&\mathit{guard}_{\texttt{redirectReq}}(e,t) \equiv \\
&\quad \exists e_{prev} \in E, m \in \texttt{Method}, u \in \texttt{Url}, hs \in \mathbb{P}(\texttt{Header}), b \in \texttt{Body} \bullet \\
&\qquad e_{prev} = \mathit{last}(t) \land \\
&\qquad \texttt{req}(m, \_, hs, b, \texttt{respMsg}(301, \{\texttt{header}(``\texttt{location}", u)\}, \_)) \in e_{prev} \land \\
&\qquad \texttt{redirectReq}(m, u, hs, b, \_) \in e
\end{aligned}$$

where $\mathit{last}(t)$ returns the last event in trace t. Note that the headers (hs) and body (b) of the original request (e) are carried over to the new, redirected request.

Customer interaction In our discussions so far, we have treated `Alice` as an abstract process that interacts solely with the store. But in reality, when the store is deployed as a web server, its customers will interact with the server through some type of HTTP client software, such as a web browser or a mobile application. To model this interaction, we will construct a process that describes the behavior of Alice using a browser to communicate to the store:

$$\texttt{AliceBrowser} = (\texttt{Alice} \underset{M_{AB}}{\parallel} \texttt{Browser})$$

[4]This classification is, by no means, complete; for example, here we are omitting details about browser scripts and AJAX requests that they may generate.

Let us assume that Alice behaves like a typical browser user, and is capable of initiating a request by interacting with the address bar or an HTML element (such as a hyperlink or form button). In particular, to add an item to her shopping cart, she would actively click on a button on a store item page, triggering a request with the item ID transmitted as a query parameter.

Recall that M_{AB} is two-part, consisting of the representation mappings for events (m_E) and values (m_V). Suppose that the designer's decision is to transmit each authentication token as a browser cookie:

$$\texttt{token}(id) \underset{m_V}{\longmapsto} \texttt{cookie}(\texttt{``authToken''}, \texttt{toString}(id))$$

The ID of the item to be added to a shopping cart is encoded as a query parameter:

$$\texttt{itemID}(id) \underset{m_V}{\longmapsto} \texttt{query}(\texttt{``item''}, \texttt{toString}(id))$$

We can then specify how the abstract `addItem` operation is to be implemented as an HTTP request as follows:

$$\begin{aligned}\texttt{Alice.addItem}(t,i) \underset{m_E}{\longleftrightarrow} \texttt{Browser.userReq}(&\texttt{GET},\\ &\texttt{url}(\texttt{origin}_{\texttt{store}}, \texttt{path}_{\texttt{addItem}}, \{sel(m_V[i] \cap \texttt{Query})\}),\\ &\{sel(m_V[t] \cap \texttt{Cookie})\}, \_, \_)\end{aligned}$$

where $sel(X)$ selects one element from set X. With this mapping, the resulting process, `AliceBrowser`, captures the expected behavior of a typical browser user. Figure 3-3 illustrates the relationship between different types of HTTP requests generated by Alice's browser. Every time Alice decides to add an item to her shopping cart, it will trigger the corresponding `userReq` in the browser. At the same time, `AliceBrowser` may also generate other requests beside those that correspond to `Store` actions; for example, Alice may visit a page on another server, or the browser may be forwarded to a particular site as a result of HTTP redirection.

Attacker Model Suppose that our malicious actor, Eve, is willing to go to great lengths to sabotage Alice's shopping cart. She deploys up her own HTTP server, and attempts to lure Alice into browsing a site that she has set up solely for the purpose of undermining the store security. Eve's machine, depicted by process `EveServer`, appears to behave like a typical web server, in that it is ready to engage in a series of `req` events. Therefore, Alice's browser may now interact not only with `StoreServer`, but with `EveServer` as well.

In order to reason about the security of the system in presence of Eve's server, we will construct a process that describes interaction between the three processes:

$$\texttt{StoreSystem} = (\texttt{AliceBrowser} \underset{M_{\texttt{Store}}}{\|} \texttt{StoreServer}) \underset{M_{\texttt{Eve}}}{\|} \texttt{EveServer}$$

Given multiple potential destination servers in the world, `AliceBrowser` determines where each request should be sent based on the hostname inside the target URL[5]. For example,

[5]In reality, the browser communicates to a DNS server to determine where a request should be sent; for

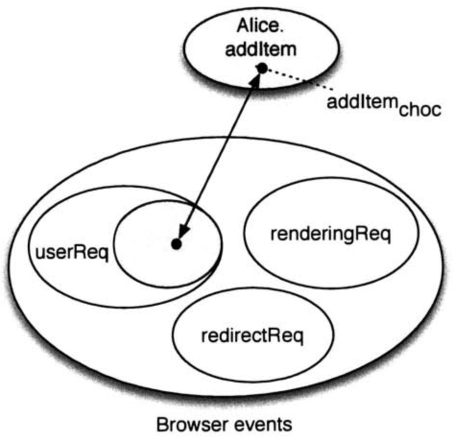


Figure 3-3: Types of HTTP requests generated by Alice's browser (process `AliceBrowser`*). The edge denotes an entry in the representation mapping. Note that the edge is bi-directional: Every* `addItem` *request initiated by Alice triggers the corresponding* `userReq` *event; conversely, every user request directed at the* `addItem` *action on the store server must have been initiated by Alice.*

every request containing `mystore.com` in its URL is sent to `StoreServer`; this communication relationship can be specified as part of $M_{\texttt{Store}}$:

$$\texttt{Browser.userReq}_{\texttt{mystore.com}} \mapsto \texttt{Server.req}$$
$$\texttt{Browser.redirectReq}_{\texttt{mystore.com}} \mapsto \texttt{Server.req}$$
$$\texttt{Browser.tagReq}_{\texttt{mystore.com}} \mapsto \texttt{Server.req}$$

where the subscript under each `Browser` request indicates the hostname of the request URL. Similarly, a requests containing `eve.com` in its URL is sent to `EveServer`; this relationship is specified in $M_{\texttt{Eve}}$ as

$$\texttt{Browser.userReq}_{\texttt{eve.com}} \mapsto \texttt{EveServer.req}$$
$$\texttt{Browser.redirectReq}_{\texttt{eve.com}} \mapsto \texttt{EveServer.req}$$
$$\texttt{Browser.tagReq}_{\texttt{eve.com}} \mapsto \texttt{EveServer.req}$$

Having established connections between various participants of the system, including the attacker, we are now ready to perform an analysis to determine whether the system satisfies the security property stated earlier. In our approach, the analysis involves finding an *attack trace* that shows how a particular sequence of events, induced by the attacker, can lead to a violation of the property. We will further discuss the details of the analysis mechanism and its automation in Chapter 4.

Attack A possible attack on the store, leading to the violation of the cart integrity, involves the following three events taking place in order (their representations are shown in Figure 3-4):

1. e_1: Alice successfully logs onto the store and receives an authentication token, which

simplicity, we will omit that detail here.

```
e_1 = {
Alice.login(aliceID, 1234, token_alice),
Browser.userReq(GET,
    url(origin_store, path_login, {query(''user'', ''aliceID''), query(''password'', ''1234'')}),
    _, _,
    respMsg(_, {setCookie(''authToken'', ''abc42'', hostname_store)}, _)),
Server.req(...),
Store.login(aliceID, 1234, token_alice)}

e_2 = {
Browser.userReq(GET,
    url(origin_eveServer, path_malicious, _), _, _,
    respMsg(_, _, html({imgTag(url_badAdd)})),
EveServer.req(...),
where url_badAdd = url(origin_store, path_addItem, {query(''item'', ''oat'')})}

e_3 = {
Browser.renderingReq(GET,
    url(origin_store, path_addItem, {query(''item'', ''oat'')}),
    {cookie(''authToken'', ''abc42'')}, _, _),
Server.req(...),
Store.addItem(token_alice, oat)}
```

Figure 3-4: Events in the CSRF attack on the store system. The notation "..." in $r(...)$ means r has the same set of parameters as the preceding representation does. The first and last events (e_1 and e_3) capture an interaction between Alice's browser and the store server; e_2 is a request where Alice visits Eve's malicious page.

is then stored as a cookie inside her browser. Based on the behavior of a standard browser, every request with a URL whose hostname matches that of the store server (`hostname`$_{\texttt{store}}$) will transmit this cookie as one of its headers.

2. e_2: Alice visits a malicious page on Eve's server, which returns an HTML document that includes an `img` tag as one of its elements. In particular, the source attribute of the tag is a URL (`url`$_{\texttt{badAdd}}$) that has been specifically crafted by Eve to point to the store server action for adding an item of her choosing (in this case, an oatmeal box).

3. e_3: As the HTML page from e_2 is rendered, the `img` tag triggers the browser to send a request at the URL specified in its `src` attribute. Since the host of the URL points to the store server, the request automatically includes the associated cookie from e_1 in its headers. The store, in turn, treats the request as a valid one coming from Alice, since it contains her authentication token. It then proceeds to insert an oatmeal box

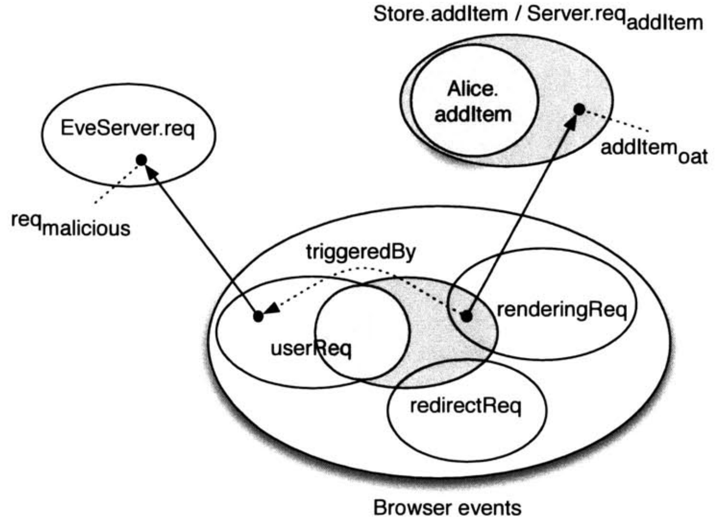


Figure 3-5: Types of HTTP requests generated by Alice's browser in presence of the store and Eve's servers (process ***StoreSystem****).*

into her shopping cart, violating the integrity property.

This attack—an instance of cross-site request forgery (CSRF)—exploits an interaction between two features of the browser: the triggering of HTTP requests without the user's initiation, and the automatic inclusion of cookies along with each request. In particular, the store becomes vulnerable to this attack as a result of two deployment decisions that specifically make use of these two features: (1) providing customer interaction through a browser, and (2) encoding an authentication token as a cookie.

Another way to explain a security failure is to articulate how certain underlying assumptions about the system may be violated when parts of the system model are elaborated with additional representations. For instance, the high-level design of the store relies on the following crucial assumption:

> The store may safely treat an `addItem` request to have originated from Alice if it includes her authentication token.

Unfortunately, this assumption is violated in the final, deployed store system, as illustrated in Figure 3-5. Although Alice would never herself initiate a request to add oatmeal to her shopping cart, her browser can be induced to send such a request to the store when she visits Eve's malicious page. This additional source of `addItem` requests, in combination with the fact that the browser blindly includes Alice's token in all requests, means that it is no longer safe for the store to assume the origin of the request based on the authentication token alone. Not surprisingly, a common prevention against CSRF attacks involves re-establishing this assumption by including an additional token (often called a CSRF token) that is transmitted to the server only under certain restricted circumstances [65].

Chapter 4

Analysis

Given a model of a system and a desired security property, the goal of an analysis is to determine whether the system satisfies the property, and if not, provide information that demonstrates how the system fails to do so. A number of different approaches may be taken to perform such an analysis. In this chapter, we describe an approach in which the analysis task is formulated as a constraint satisfaction problem, and discuss its potential benefits and drawbacks over other techniques.

Chapter Highlights

- An analysis task is formulated as the problem of finding a counterexample trace to a given safety property (Section 4.1)
- Given a *partial* specification of representation mappings, the same analysis technique can be used to generate complete mappings that may introduce vulnerabilities into the resulting system (4.2).
- An attacker is modeled as a process that may engage in any of the system events without being required to adhere to their guards (4.3.1).
- Certain types of data may be considered more sensitive than others; our analysis requires them to be explicitly specified by the designer (4.3.2).
- The overall analysis problem can be tackled automatically using a finite-domain constraint solver (4.4).

4.1 Analysis as Counterexample Detection

Although the examples that we have discussed so far are security-related, our framework, in general, allows an analysis of *safety properties* over traces [48]. Informally, a safety property states that a certain "bad thing" should never occur during the system execution. In our approach, a property is expressed in the form of

$$Prop(t)$$

where *Prop* is a predicate, parameterized over trace *t*, that characterizes what it means for *t* to be safe. The goal of an analysis is then to check whether this predicate holds over every possible behavior of the system:

$$\forall t \in \mathit{traces}(\mathit{Sys}) \bullet \mathit{Prop}(t)$$

where *Sys* is the process that represents the entire system. Instead of checking that every trace satisfies *Prop*, we will instead reformulate the analysis problem into the task of finding a *counterexample trace* that demonstrates how the property may be violated. This is done by taking the negation of *Prop* and attempting to find a witness trace t to the resulting formula:

$$\exists t \in \mathit{traces}(\mathit{Sys}) \bullet \neg \mathit{Prop}(t)$$

If no such counterexample trace exists, then we may conclude that the system indeed satisfies the given property.

Example Recall, from Section 3.3.1, the property that describes the integrity of shopping carts on the online store:

> *A shopping cart should only contain items that its owner intends to purchase.*

which can be formalized as the following predicate over trace t:

$$\begin{aligned}&\texttt{CartIntegrity(t)} \equiv\\ &\quad \forall i \in \texttt{ItemID}, c \in \texttt{Cart}, p \in \texttt{Customer} \bullet\\ &\qquad c = \texttt{getCart}(\texttt{username}(p), t) \land i \in c.\mathit{items} \Rightarrow i \in \texttt{shoplist}(p)\end{aligned}$$

where `Customer` is the set of customer processes in the system, and `username(p)` is an auxiliary function that returns the username associated with customer p.

Our analysis then involves finding a witness trace t to the negation of `cartIntegrity`:

$$\begin{aligned}&\exists t \in \mathit{traces}(\texttt{StoreSystem}) \bullet\\ &\quad \exists i \in \texttt{ItemID}, c \in \texttt{Cart}, p \in \texttt{Customer} \bullet\\ &\qquad c = \texttt{getCart}(\texttt{username}(p), t) \land i \in c.\mathit{items} \land i \notin \texttt{shoplist}(p)\end{aligned}$$

A counterexample, if it exists, would describe a possible system execution (t) in which the shopping cart (c) of a customer (p) contains an item (i) that she does not intend to purchase.

4.2 Representation Mapping Specification and Generation

In the previous two chapters, we introduced the notion of a representation mapping, $m_E \subseteq R_E \times R_E$, where

$$(a, b) \in m_E$$

means that every event labeled a is to be assigned b as an additional representation during the composition of a pair of processes (similarly, $m_V \subseteq R_V \times R_V$ for data representations).

In practice, it would be quite tedious to specify a representation mapping by explicitly listing all of its entries. An alternative and more viable approach is to instead specify the mapping *declaratively* in the following style:

$$S = \{a, b \in R_E \mid \; C(a,b)\}$$

where C is a predicate that describes a relationship between the parameters of a and b; we will call this set comprehension a *mapping specification* (S). A candidate mapping m_E *satisfies* a specification S if C evaluates to true over every tuple in m_E; in other words,

$$m_E \subseteq S$$

We have already seen an example of this style of mapping in Section 3.3.1, where we described how the abstract `addItem` operation may be implemented as an HTTP request:

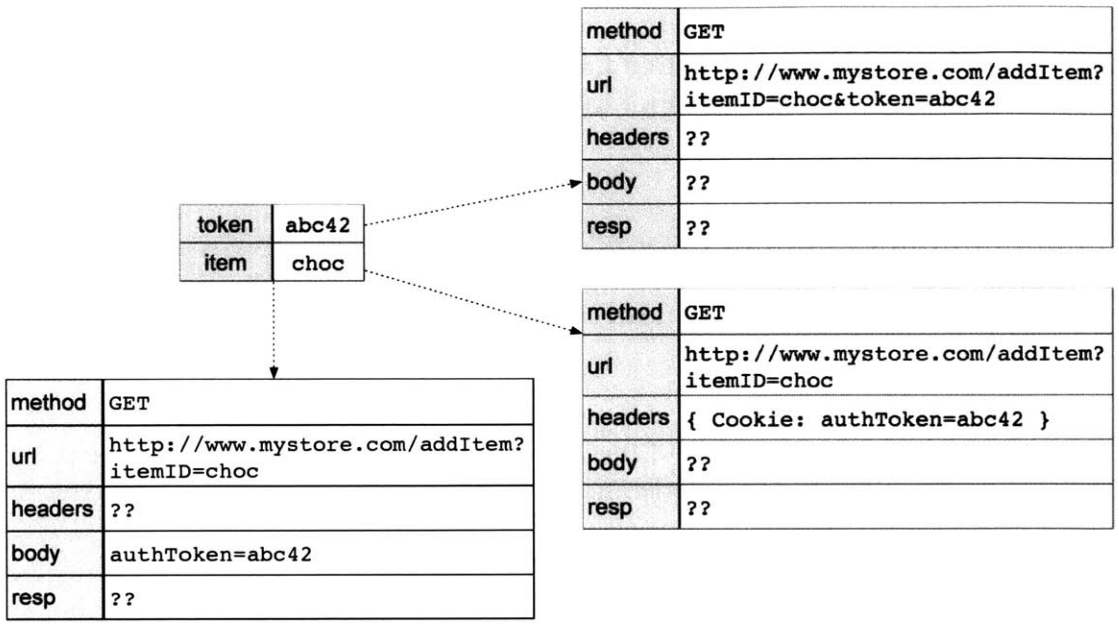
$$\begin{aligned}
&\{\texttt{addItem}(\mathit{token}, \mathit{item}), \texttt{userReq}(\mathit{method}, \mathit{url}, \mathit{headers}, \mathit{body}, \mathit{resp}) \in R_E \mid \\
&\quad \mathit{method} = \texttt{GET} \wedge \\
&\quad \mathit{url} = \texttt{url}(\texttt{origin}_{\texttt{store}}, \texttt{path}_{\texttt{addItem}}, \{\texttt{query}(\text{``}\texttt{item}\text{''}, \texttt{toString}(\mathit{item.id}))\}) \wedge \\
&\quad \mathit{headers} = \{\texttt{cookie}(\text{``}\texttt{authToken}\text{''}, \texttt{toString}(\mathit{token.id}))\} \wedge \\
&\quad \exists b \in \texttt{Body} \bullet \mathit{body} = b \wedge \exists \mathit{rm} \in \texttt{RespMsg} \bullet \mathit{resp} = \mathit{rm}\}
\end{aligned}$$

According to this mapping, the item ID is to be encoded as a URL query parameter, and the authentication token is to be transmitted as a cookie. Here, we are not concerned with the body or the response of the request, and so they are simply assigned arbitrary values (b and rm) that have no relation to the parameters of `addItem`.

Partial mapping specification An important consequence of the declarative approach is that a mapping may be specified only *partially*. For instance, in an alternative specification of the `addItem` mapping, we may leave unspecified how the authentication token is to be transmitted as part of the HTTP request:

$$\begin{aligned}
&\{\texttt{addItem}(\mathit{token}, \mathit{item}), \texttt{userReq}(\mathit{method}, \mathit{url}, \mathit{headers}, \mathit{body}, \mathit{resp}) \in R_E \mid \\
&\quad \mathit{method} = \texttt{GET} \wedge \\
&\quad \exists \mathit{qs} \in \mathbb{P}(\texttt{Query}) \bullet (\mathit{url} = \texttt{url}(\texttt{origin}_{\texttt{store}}, \texttt{path}_{\texttt{addItem}}, \mathit{qs}) \wedge \\
&\qquad\qquad \texttt{query}(\text{``}\texttt{item}\text{''}, \texttt{toString}(\mathit{item.id})) \in \mathit{qs}) \wedge \\
&\quad \exists \mathit{hs} \in \mathbb{P}(\texttt{Header}) \bullet \mathit{headers} = \mathit{hs} \wedge \\
&\quad \exists b \in \texttt{Body} \bullet \mathit{body} = b \wedge \exists \mathit{rm} \in \texttt{RespMsg} \bullet \mathit{resp} = \mathit{rm}\}
\end{aligned}$$

We can further simplify this expression by removing statements about the request parameters (*headers*, *body*, and *resp*) that are not explicitly related to `addItem`:



token
abc42
item
choc
method
GET
url
http://www.mystore.com/addItem?itemID=choc&token=abc42
headers
??
body
??
resp
??
method
GET
url
http://www.mystore.com/addItem?itemID=choc
headers
{ Cookie: authToken=abc42 }
body
??
resp
??
method
GET
url
http://www.mystore.com/addItem?itemID=choc
headers
??
body
authToken=abc42
resp
??

Figure 4-1: Three possible encodings of `addItem` *as an HTTP request, satisfying the partial mapping specification.*

$$
\begin{aligned}
\{&\texttt{addItem}(\mathit{token}, \mathit{item}), \texttt{userReq}(\mathit{method}, \mathit{url}, \mathit{headers}, \mathit{body}, \mathit{resp}) \in R_E \mid \\
&\quad \mathit{method} = \texttt{GET} \wedge \\
&\quad \exists\, qs \in \mathbb{P}(\texttt{Query}) \bullet \mathit{url} = \texttt{url}(\texttt{origin}_{\texttt{store}}, \texttt{path}_{\texttt{addItem}}, qs) \wedge \\
&\qquad\qquad\qquad\qquad \texttt{query}(\text{``}\texttt{item}\text{''}, \texttt{toString}(\mathit{item.id})) \in qs\}
\end{aligned}
$$

This specification states that the item ID is to be encoded as a query parameter, but does not say anything about the authentication token. Semantically, the specification allows all possible ways of mapping the token into an HTTP request; it may be carried as a cookie, a query parameter, or as part of the request body, as shown in Figure 4-1.

Property-guided mapping generation In the space of possible mappings that satisfy a given specification, ones that are of particular importance from the analysis perspective are those that may admit "unsafe" traces in the resulting system. Intuitively, these mappings describe *insecure* design decisions, in that they may introduce behavior that can potentially be exploited for an attack.

Given a partial specification of a mapping, a single analysis can be used to not only find a counterexample trace that violates a given property, but also generate a complete mapping that permits such a trace to be a valid behavior of the resulting system. To be more precise, let $p_1, p_2, ..., p_n$ be the set of n processes in the system, and $S_1, S_2, ..., S_{n-1}$ be the set of user-specified mapping specifications, where S_k specifies the relationship between processes p_k and p_{k+1}. Then, the *mapping generation* problem can be stated as finding witnesses to the following existential formula:

$$\begin{aligned}
&\exists_{i=1}^{n-1} M_i \in M \bullet (\bigwedge_{j=1}^{n-1} sat(M_j, S_j)) \land \\
&\quad \exists\, Sys \in P \bullet Sys = compose(\{p_1, p_2, ..., p_n\}, \{M_1, M_2, ...M_{n-1}\}) \land \\
&\qquad \exists\, t \in traces(Sys) \bullet \neg Prop(t)
\end{aligned}$$

where $sat(M_j, S_j)$ evaluates to true if and only if M_j satisfies the specification S_j, and $compose(X, M)$ returns a process that results from the pairwise composition of the processes in X using the set of mappings M:

$$compose(\{p_1, p_2, ..., p_n\}, \{M_1, M_2, ...M_{n-1}\}) = (((p_1 \underset{M_1}{||} p_2) \underset{M_2}{||} p_3) \underset{M_3}{||} ...) \underset{M_{n-1}}{||} p_n$$

Informally, the new analysis problem involves generating a set of representation mappings that (1) satisfy the user-specified specifications, and (2) when used in the composition of processes, allow the resulting system to produce a trace that leads to the violation of a given property.

Example Consider the composition of `Alice` and `Browser` using mapping M_{AB}:

$$\texttt{Alice} \underset{M_{AB}}{||} \texttt{Browser}$$

Since M_{AB} is two-part (m_E and m_V for events and data), we will also need to specify a pair of specifications, S_E and S_V. We earlier specified the mapping between `addItem` and `userReq`, requiring that the *item* parameter of the abstract request to be encoded as a query parameter, but not stating anything explicit about `token`:

$$\begin{aligned}
S_E = \{&\texttt{addItem}(token, item), \texttt{userReq}(method, url, headers, body, resp) \in R_E \mid \\
&method = \texttt{GET} \land \\
&\exists\, qs \in \mathbb{P}(\texttt{Query}) \bullet url = \texttt{url}(\texttt{origin}_{\texttt{store}}, \texttt{path}_{\texttt{addItem}}, qs) \land \\
&\qquad \texttt{query}(\texttt{``item''}, \texttt{toString}(item.id)) \in qs\}
\end{aligned}$$

We will leave the mapping between data representations completely unconstrained:

$$S_V = \{a, b \in R_V \mid \texttt{True}\}$$

Essentially, this specification allows the analysis to explore all possible ways of mapping the store-level data types (`ItemID` and `AuthToken`) into HTTP-level ones (`Query`, `Cookie`, etc.,), with one caveat: Since S_E requires that *item* be encoded as a query parameter, the analysis will only explore candidate mappings where `ItemID` is mapped to `Query`.

Given the specifications (S_E, S_V), the analysis attempts to generate a pair of mappings (m_E, m_V) such that

$$m_E \subseteq S_E \land m_V \subseteq S_V \land \exists\, t \in traces(\texttt{StoreSystem}) \bullet \neg\texttt{CartIntegrity}(t)$$

Intuitively, the generated mappings describe insecure deployment decisions that may introduce vulnerabilities into the store, allowing an attacker to undermine the integrity of the shopping carts. For instance, given a choice of possible mappings, the analysis may select one where Alice's token is encoded as a cookie and transmitted as part of a HTTP request header:

$$\begin{aligned} &(\mathtt{addItem}(\mathtt{token}_{\mathtt{alice}}, \mathtt{choc}),\\ &\ \mathtt{userReq}(\mathtt{GET}, \mathtt{url}(\mathtt{origin}_{\mathtt{store}}, \mathtt{path}_{\mathtt{addItem}}, \{\mathtt{query}(``\mathtt{item}", ``\mathtt{choc}")\}),\\ &\qquad \{\mathtt{cookie}(``\mathtt{authToken}", ``\mathtt{abc42}")\}, \{\}, \mathtt{rm})) \in m_E \end{aligned}$$

where *rm* is some arbitrary response message. In Section 3.3.1, we have already seen how the decision to use cookies to transmit authentication tokens can leave the system vulnerable to CSRF attacks. In fact, the witness trace t that the analysis generates here, along with this mapping, may be the same sequence of events in Figure 3-4. Given this feedback from the analysis, the designer may attempt to repair the resulting model to prevent this attack (e.g., add a CSRF protection token) or explore alternative encoding of the token by modifying the mapping specification.

This ability to explore system behaviors with only partially specified mappings is especially beneficial during early design stages. At this point, some design choices may be yet to be made, and so specifying a complete mapping would simply be an impossible task. By leveraging our analysis, the designer can explore a space of design decisions and identify those that may introduce vulnerabilities into the system—thus, being able to address them before committing to an implementation.

4.3 Security Considerations

4.3.1 Threat Model

An important part of any security analysis is a *threat model*—a description of malicious agents in the environment and their capabilities.

A common approach to defining a threat model involves explicitly enumerating a list of actions that the attacker may take in order to undermine the security of a system [77, 86]. For example, a threat model used for an analysis of a web application may include statements such as "the attacker may attempt to spoof another user's HTTP request" or "the attacker may trick a user into navigating to a malicious page". There are two potential downsides to this approach. First, manually attempting to devise an exhaustive list of potential threats can be a tedious and error-prone task. Second, statements such as the ones above may be too generic, and an extra step may be needed to instantiate them against a particular system model.

In our approach, a threat model is treated as an implicit part of the system model. **An attacker is like any other process, except that it behaves in an unconstrained manner**, in that it does not adhere to the guard conditions that are normally associated with a process of the same type. For example, the guards assigned to the `Browser` process describe the expected behavior of a standard browser (e.g., it should send only those cookies that are associated with the origin of an HTTP request). An attacker acting as a `Browser` process, however, may choose not to follow these rules; imagine, for example, a hacker who modifies the source code of the browser to generate any HTTP requests as desired. **In other words, any assumptions made about the behavior of a process are discarded when the process is considered to be malicious.**

More formally, during analysis, a set of processes are designated to be untrusted:

$$Untrusted \subseteq P$$

The overall behavior of each untrusted process $u \in Untrusted$ is defined as follows:

$$\begin{aligned} traces(p) = \{t \in Trace \mid t = \langle\rangle \vee \\ \exists t' \in traces(p), e \in E \bullet t = t' \frown \langle e \rangle \wedge e \in dom(rep)\} \end{aligned}$$

Note that this definition does not even mention the guards; the process is free to perform any event e as long as it is assigned at least one representation.

However, even untrusted processes must still adhere to the dataflow axiom that was discussed in Section 3.1; it cannot access a piece of data unless it receives that value as part of an event, or already owns the value at the beginning of the execution. Without this axiom, the analysis may generate bogus counterexamples where an attacker is able to, for example, magically guess the password of a user and trivially cause a security violation.

4.3.2 Data Classification

In security, not all values are considered equal. A piece of string representing a user's credential, for example, may be considered more private than a string that represents a username. A typical security analysis relies on an assumption that these private values are not accessible to malicious processes at the beginning of the system execution; without this assumption, the analysis would generate counterexamples where a security property of the system is trivially compromised.

In our analysis approach, some subset of data representations are designated to be *private*:

$$Private \subseteq R_V$$

The following axiom ensures that the set of values initially available to an untrusted process is limited to non-private ones:

$$\forall u \in Untrusted \bullet \neg(\exists v \in V, r \in v \bullet v \in owns(u) \wedge r \in Private)$$

The content of *Private* is specified by the designer for a particular system model. In the store system, Alice's password and authentication token are considered critical for security; if the attacker is able to access either one of the two, she would be able to sabotage Alice's account by simply pretending to be her. The following can be specified by the designer to ensure that these two are initially inaccessible to the attacker:

$$Private = \{\texttt{token}_{\texttt{alice}}, \texttt{password}_{\texttt{alice}}\}$$

Of course, this does not necessarily imply that this information will always be kept private; for example, the attacker may attempt to gain access to Alice's credential through a phishing or man-in-the-middle attack.

Sometimes, it may not be immediately obvious to the designer which information should be considered private. In that case, `Private` may simply be specified to be empty:

$$Private = \emptyset$$

For most systems, an ensuing analysis would likely generate a counterexample that demonstrates how the attacker leverages its initial knowledge to compromise the system. The designer can examine such counterexamples and incrementally refine the specification of `Private` over multiple analysis phases, eventually arriving at a reasonable assumption about the attacker's knowledge.

4.3.3 Cryptography

A security analysis typically involves reasoning about cryptographic operations. To securely transmit a piece of data to its peers, a process may encrypt the data into a *ciphertext*, which can only be decrypted back into its original *plaintext* using a designated *key*. Most protocol languages such as the spi calculus [2] have built-in primitives for expressing these operations. In our approach, no built-in mechanism is necessary.

One way to model these operations in our approach is to construct a process that acts like a cryptography library, providing services for both encrypting and decrypting a piece of data. For example, consider a process named `SymmCrypto`, which represents a library for symmetric-key cryptography. It engages in two types of events:

- `encrypt`(*plain*, *key*, *cipher*): Takes a piece of plaintext (*plain* ∈ `Plaintext`) and a key (*key* ∈ `Key`), and returns the ciphertext (*cipher* ∈ `Ciphertext`) that results from encrypting the plaintext with that key.
- `decrypt`(*cipher*, *key*, *plain*): Decrypts the ciphertext with the key and returns the resulting plaintext.

How do we ensure that the decryption operation returns the correct corresponding ciphertext for a given plaintext? A relationship between a pair of cipher and plaintext can be expressed as a guard condition on `decrypt` events, as follows:

$$\begin{aligned}&guard_{\texttt{decrypt}}(e,t) \equiv\\&\quad \exists e_{enc} \in E, p \in \texttt{Plaintext}, k \in \texttt{Key}, c \in \texttt{Ciphertext} \bullet in(e_{enc},t) \land\\&\qquad \texttt{encrypt}(p,k,c) \in e_{enc} \land \texttt{decrypt}(c,k,p) \in e\end{aligned}$$

where $in(e,t)$ means that e is an event belonging to trace t. This guard stipulates that the result of decryption must be the plaintext that was previously encrypted with the same key to produce the given ciphertext.

We must also ensure that (1) the encryption operation is a function, in that it returns exactly one ciphertext for each pair of a plaintext and a key, and (2) it is not possible to decrypt a ciphertext back to the same plaintext using two different keys. These conditions can be expressed as a guard on encrypt events:

$$
\begin{aligned}
& guard_{\texttt{encrypt}}(e,t) \equiv \\
& \quad \exists p \in \texttt{Plaintext}, k \in \texttt{Key}, c \in \texttt{Ciphertext} \bullet \\
& \quad\quad \texttt{encrypt}(p,k,c) \in e \land \\
& \quad\quad \neg \exists e' \in E, k' \in \texttt{Key}, c' \in \texttt{Ciphertext} \bullet \\
& \quad\quad\quad in(e',t) \land \texttt{encrypt}(p,k',c') \in e' \land \\
& \quad\quad\quad ((k = k' \land c \neq c') \lor (k \neq k' \land c = c'))
\end{aligned}
$$

Informally, this guard states that (1) encrypting a piece of plaintext p with key k should always yield the same ciphertext (left side of the disjunction in the last line), and (2) no two different keys can be used to encrypt p into c (right side of the disjunction).

There are two advantages of explicitly modeling cryptographic operations as we have done, instead of providing them as built-in primitives. First, adding another type of cryptographic operations (e.g., public-key cryptography) requires no change to our underlying modeling formalism or analysis technique. Second, this approach can be used to reason about different *implementations* of cryptographic operations. We may, for example, take the above *abstract* model of symmetric-key encryption and then relate it to a concrete implementation model using a representation mapping; this would allow us to check whether security guarantees provided by the abstract model is retained at a lower-level of abstraction (as it turns out, there are a number of cryptographic algorithms that have been theoretically proven to be secure but are still vulnerable to attacks when deployed in real settings [10, Chapter 3]).

4.4 Constraint Formulation

4.4.1 Analysis Problem

In summary, the overall analysis problem can be formulated as finding a set of witnesses that satisfy the following formula:

$$
\begin{aligned}
& \overset{n-1}{\underset{i=1}{\exists}} M_i \in M \bullet \Big(\bigwedge_{j=1}^{n-1} sat(M_j, S_j) \Big) \land \\
& \quad \exists\ attackers \subseteq \mathbb{P}(P), init \subseteq P \times V \bullet \\
& \quad\quad Untrusted = attackers \land owns = init \land \\
& \quad\quad \exists Sys \in P \bullet Sys = compose(\{p_1, p_2, ..., p_n\}, \{M_1, M_2, ...M_{n-1}\}) \land \\
& \quad\quad\quad \exists t \in traces(Sys) \ \bullet \neg Prop(t)
\end{aligned}
$$

The nesting of quantifiers illustrates different types of system parameters that are explored during the analysis. Informally, the analysis task can be summarized as (1) finding an assignment of representations to events and data values where (2) some subset of processes are designated as untrusted, and the processes are initially granted access to certain pieces of information such that (3) the resulting system permits an execution that leads to a violation of the given property.

4.4.2 Finalization

In general, proving or refuting a first-order logic formula, like the one shown above, is an undecidable problem. To support an automated analysis with a termination guarantee, we transform the analysis task into a constraint solving problem over **finite domains**. This step involves finitizing the size of the universe in a given model, placing a bound on (1) the total number of process, events and data values, and (2) the maximum length of traces generated by the processes.

Once the domains have been finitized, the resulting constraint can be handed off to a finite model finder, which will attempt to find a satisfying instance to a given formula. In the next chapter, we will discuss a prototype implementation that uses a first-order relational modeling tool called the Alloy Analyzer [42], which itself relies on a model finder called Kodkod [82].

4.5 Limitations

There are certain types of security properties, such as non-interference in information flow, that cannot be captured as a safety property [54]. These belong to a general class of properties called *hyperproperties* [23], which are statements about the behavior of a system across *multiple traces*, instead of over a single trace. Specifying a hyperproperty lies outside the realm of a first-order logic, as it involves quantification over sets (in particular, sets of traces), and an ensuing analysis also requires higher-order reasoning.

The security guarantee provided by our analysis is only up to the user-provided bounds; even if the analysis fails to find a counterexample, there may exist a violating trace that involves a larger number of events or entities. We believe that this is an acceptable compromise to achieve automation. In practice, many security flaws can be demonstrated with a small number of objects [11]. If greater confidence is desired, the model can be re-analyzed with larger bounds as necessary; among the systems that we have analyzed, we have not yet seen any counterexample that involved more than 12 events or entities.

Most cryptographic algorithms provide *probabilistic*, rather than absolute, guarantees about the secrecy of encrypted data. The underlying logic in our framework is not suitable for expressing and reasoning about such probabilistic properties, and so our approach to modeling cryptography may be considered somewhat imprecise. In our experience, however, we have found that many security properties can be specified without probabilistic expressions, and our approach is adequate in most situations.

Chapter 5

Implementation

This chapter presents *Poirot*[1], a prototype implementation of our conceptual modeling and analysis framework. We describe three major aspects of the tool, and challenges involved in implementing them: (1) a language for specifying system models, (2) the translation to the Alloy modeling language, and (3) the analysis of an input model against a security property.

5.1 Overview

The high-level architecture of Poirot is shown in Figure 5-1. Our user—a system designer or a security analyst—interacts with the tool by specifying a model of a system, along with its desired security properties, in the input language of Poirot. The user may additionally provide representation mappings that describe the relationship between the system model and one or more domain models in Poirot's library.

Given the input system model, representation mappings, and relevant domain models, Poirot compiles them into a single, global model in an intermediate specification language called Alloy [42]. Its analysis tool, the Alloy Analyzer, is then used to automatically check the model against a desired security property; a counterexample trace, if found, demonstrates a violation of the property as a sequence of events.

Poirot is intended to be used in an *incremental* manner. Starting with an initial model that describes a high-level design of the system, the user may elaborate various parts of the model with a choice of representation, gradually transforming the model into a more detailed one. The tool also facilitates an *exploration* of design alternatives; by leaving one or more mappings partially specified, the user can ask the tool to identify candidate mappings and explain why they may be problematic for security through concrete examples. This approach is especially useful during early development stages, where some design decisions may be unknown, and where system flaws can still be addressed at a relatively low cost.

[1]Agatha Christie's *Murder on the Oriental Express* is a story of detective Hercules Poirot's investigation of a mysterious murder on a passenger train. In the novel, he meticulously interviews all potential suspects on the train, each interview revealing new (and possibly overlapping) details about the circumstances of the crime. Through this process, Poirot incrementally constructs a comprehensive picture of the crime, and works towards identifying the guilty.

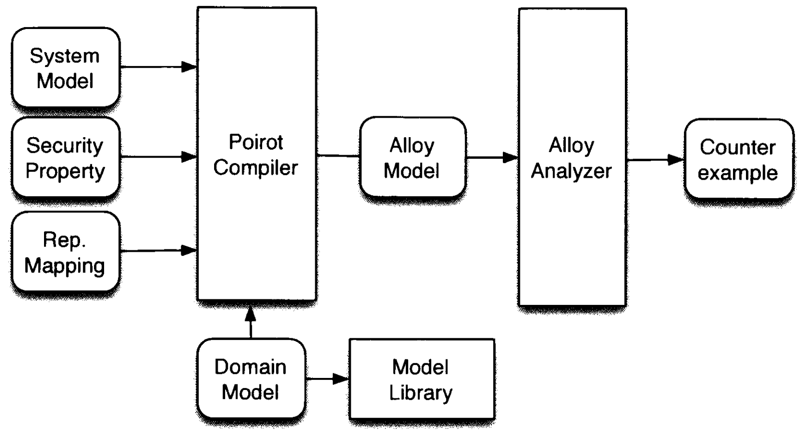


Figure 5-1: Architecture of Poirot.

5.2 Modeling Language

Poirot provides its own input language for specifying a model of the system, its desirable properties, and mappings between models. The language has been implemented as an embedded DSL in Ruby, which provides a flexible metaprogramming facility for building DSLs. In particular, our DSL uses aRby [55], a library for embedding declarative, relational constraints inside a Ruby program, with capability to translate the program into an Alloy model.

System modeling Figure 5-2 shows a model of a simple online store system in Poirot. A typical Poirot model consists of a set of of *data types*, *components*, and *operations*.

Each *data type* represents a set of data values, and may contain one or more *fields*. For example, `Username` is a primitive data type that represents a set of usernames, whereas `ItemInfo` is a composite type that consists of two named fields, `name` and `price`. As discussed in Section 3.2, one or more data types may be declared to be *private*, meaning they are inaccessible to malicious processes at the beginning of the system execution. In this model, `Token` and `Password` are important pieces of data used to control access to shopping carts and thus, declared to be private.

A *component* defines the structure and behavior of a group of processes, and contains a set of its own *fields* and *operations*. By default, component fields are *static*, in that their values are fixed throughout the system execution. In this particular store model, we will assume that the information about the items (`catalog`) and the user passwords will remain fixed (`passwords`)[2]. Some of the fields may be declared to be dynamic by using the keyword `updatable`. The set of authentication tokens assigned to customers, as well as the content of their shopping carts, may evolve over time as the store performs its services, and so are modeled as dynamic fields (lines 14-15).

Following the field declarations is a list of component *operations*, which describe different types of events that are performed by this component. Each operation consists of a set

[2]The expression `a**b` represents a binary relation from data type `a` to `b`; for example, `passwords` can be regarded as a record of usernames and their passwords.

```
# Datatype declarations
data Username, ItemID, ItemName, Price
data ItemInfo[name: ItemName, price: Price]
data Cart[items: (set ItemID)]
privat data Password  # user password
privat data Token  # authentication token

# component declarations
component Store [
  # component states are static by default
  catalog: ItemID ** ItemInfo,
  passwords: Username ** Password,
  # updatable states
  tokens: (updatable Username ** Token),
  carts: (updatable (set Username) ** (set ItemID)),
]{
  # operation declarations
  # by convetion, return data is named 'ret'
  op Login[u: Username, p: Password, ret: Token] {
    # guard
    ensures { passwords[u] == p }
    # stateful operation; assign a new token 't' to user
    updates { make(t: Token) { tokens.insert(u ** t) and ret == t }}
  }
  # adds a new item to the shopping cart of the user identified by 't'
  op AddItem[t: Token, i: ItemID] {
    updates { carts.insert(tokens.(u) ** i) }
  }
  # returns the content of the shopping cart identified by 't'
  op DisplayCart[t: Token, ret: Cart] {
    ensures { make (c: Cart) { c.items == carts[tokens.(u)] and ret == c }}
  }
  # returns the information about the item 'i'
  op GetItem[i: ItemID, ret: ItemInfo] {
    ensures { ret == catalog[i] }
  }
}
component Customer [
  id: Username,
  # a set of items that customer wishes to purchase
  wishlist: (set ItemID)
]{
  # 'invokes' declares a type of output events that customer engages in
  invokes { Store::Login }
  # output event with a guard
  invokes { Store::AddItem.onlyIf {|o| o.(i).in?(wishlist)} }
  invokes { Store::DisplayCart }
  invokes { Store::GetItem }
}

property cartIntegrity {
  all(s: Store, u: Username, i: Item) {
    customer = id.(u)
    # item must be in customer's wishlist if it's been added to her cart
    i.in? customer.wishlist if contains(s.carts, u ** i)
  }
}
```

Figure 5-2: A model of an online store system in Poirot.

of argument and return parameters, guard conditions (`ensures`) and update statements (`updates`). For instance, given username `u` and password `p`, the login operation checks whether `p` matches that of the user `u` in its `passwords` record (line 21); if so, it allocates a new token t for u and updates its `tokens` field before returning the token back to the user (line 23)[3]. If the input parameters of an operation do not satisfy a guard condition, the operation is considered a failure and prevented from taking place. For example, if an incorrect password is provided for a login request (i.e., the condition on line 21 is violated), then the store will simply refuse to participate in the operation.

Each customer, as defined by the component `Customer`, has an username (`id`) that she uses to identify herself to the store, and a list of items that she wishes to purchase (`wishlist`). The customer interacts with the store by *invoking* one or more operations that are provided by the latter. Similar to an `ensures` statement inside an operation, we may also define a condition under which a component is allowed to invoke a particular operation. For instance, we may want to assume that a customer will never send a request to add an item that is not on her wishlist; this is expressed by attaching a condition to the invocation of `AddItem` (line 46), requiring that the item parameter (`i`) in each request (`o`) must always be a member of `wishlist`[4].

Property A property of a model is specified as a logical formula inside the construct `property`, and may talk about data values that are accessible to a process or the content of a particular component field during the system execution, by using the following built-in constructs:

- `mayAccess`(p, v): Evaluates to true if process p is able to access data value v at any point in the execution.
- `contains`(f, t): Evaluates to true if component field f contains tuple t at any point in the execution.

For example, the property `cartIntegrity` says that if item i has been added to the shopping cart owned by user u at some point in the execution, then that item must belong to the user's wishlist (lines 52-55).

Mapping As discussed in Section 4.2, a representation mapping is specified as a declarative constraint that relates the structures of a pair of operations or data types. Suppose that the store operations are to be implemented as HTTP requests. Figure 5-3 shows a part of a model that describes a generic HTTP server and its related datatypes, and Figure 5-4 shows examples of the mappings between the store and HTTP models.

The mapping `addItemToReq` describes how each `AddItem` event (a), performed by store s, is encoded as an HTTP request (h) served by r. In particular, the mapping states that authentication token and the item ID are to be transmitted as a header and a query parameter of the request, respectively. We also want to ensure that all HTTP requests implementing `AddItem` have a common URL prefix (e.g., `http://www.mystore.com/addItem`). To achieve

[3]The keyword `f.insert(t)` modifies the content of the field `f` by adding a new tuple `t`.

[4]The expression `a.in?(S)` means `a` belongs to set `S`.

```
# Data types related to HTTP
data Host, Path, Resource
data URL[host: Host, path: Path, query: (set Query)]
data Query[name: String, val: String]
data Header[name: String, val: String]
data Cookie < Header  # Cookie is a subtype of Header

data Body[data: (set String)]
data RespMsg[headers: (set Header), res: Resource]

# HTTP server
component Server [
 resources: (updatable URL ** Resource),
 hostname: Host
]{
  op HttpReq[url: URL, headers: (set Header), body: Body, ret: RespMsg]{
    # the host of the URL must match this server's hostname
    ensures { url.host == hostname }
    # return the resource associated with the URL
    ensures { ret.res == resources[url] }
  }
}
```

Figure 5-3: Part of an HTTP model in Poirot.

this, we will specify that every request URL is constructed from the hostname of the server *r*, and assigned a unique, fixed `Path` value (called `PathAddItem`, introduced on line 1[5]).

The encoding of an authentication token as a cookie is specified in a similar style using the mapping `tokenToCookie`. Every cookie that encodes an authentication token is given a fixed name, which is represented by a string value called `StrAuthToken` (line 2). The value of the cookie is assigned a string representation of the token; to do this, we use a built-in function called *toStr*, which converts a given data value to a unique, string constant.

5.3 Translation to Alloy

A model of a system specified in Poirot is translated into Alloy [42], a modeling language based on a first-order relational logic with transitive closure. We found three features of Alloy particularly useful as the target language for Poirot: (1) its flexible type system, which allows an element to be associated with multiple types (crucial for our composition mechanism), (2) its partial, declarative nature, which allows representation mappings to be specified only partially, and (3) its automated analysis engine, with its ability to generate counterexamples.

5.3.1 Encoding of the Basic Modeling Concepts

Figure 5-5 shows an Alloy model that provides an encoding of the fundamental concepts from our modeling approach, such as events, traces, and dataflow between processes. Every Alloy model translated from Poirot relies on the concepts in this model.

[5]The keyword **`one`** introduces a data type that has exactly one element in it.

```
one data PathAddItem < Path
one data StrAuthToken < Str

mapping addItemToReq[s: Store, a: AddItem, r: Server, h: HttpReq] {
  a.t.in? h.headers and         # token is mapped to a header
  a.i.in? h.url.query and       # item is transmitted as a query parameter
  h.url.host == r.hostname and    # URL host is set to the server hostname
  h.url.path == PathAddItem       # path is fixed to a constant
}

mapping tokenToCookie[t: Token, c: Cookie] {
  c.name == StrAuthToken and          # cookie name is fixed to a constant
  c.val == toStr(t)      # toStr(t) converts t to a unique string constant
}
```

Figure 5-4: Examples of mappings between the store and HTTP models.

In our approach, a system consists of three basic types of entities: data values, events, and processes, represented by signatures `Data`, `Event`, and `Proc`, respectively[6]. Each data value may contain one or more other values (represented by `flds` on line 4); given a particular value `v`, the function `reachable` computes the set of all values that are reachable from `v` through `flds`[7]. It would be rather strange for a value to contain itself, and so a constraint is added to ensure that no value is reachable from itself (line 6)[8].

Every event is associated with a pair of sender and receiver processes, and may additionally carry a set of argument and return values (lines 14-17). A signature constraint is added to ensure that a process cannot send a message to itself (line 19). The last two constraints encode dataflow axioms (lines 20-22); we will return to these shortly.

In Chapter 2, we described how the overall behavior of a system can be modeled as a set of event traces that it allows. In our encoding, a trace is represented implicitly as a set of totally ordered `Event` atoms (line 1)[9]. Conceptually, each `Event` atom corresponds to a particular point in the system execution, and can be used as an index to retrieve the snapshot of the system state at that point; we will see an example of this modeling idiom in the next section.

A process starts out with a set of data values that it owns (line 26), and gains access to other data as it further sends or receives events (line 27). The relation `mayAccess` keeps track of the set of data values that may be accessible to processes at different points over the system execution. More precisely, if (p, v, e) is a tuple in `mayAccess`, it means that value v may be accessible to process p when event e is about to take place.

The signature constraint for `Proc` is an axiom that describes different ways in which process p may come to access a particular data value, v:

[6]The Alloy keyword `sig` introduces a *signature*, which corresponds to a set of atoms in the universe. Each signature may contain one or more *fields*, each introducing a relation that maps the elements of the signature to the field express; for example, `sender` in `Event` is a binary relation that maps each `Event` atom to the `Proc` atom that represents its sender.

[7]The operator ^ computes a transitive closure of a relation.

[8]The block of field declarations inside a signature may be followed by a set of *signature constraints*, which are imposed on every member of that signature.

[9]The built-in `ordering` library imposes a total order on the elements of a given signature.

```
open util/ordering[Event]

sig Data {
  flds : set Data          -- fields
}{
  this not in reachable[this]
}

fun reachable[v : Data] : set Data {
  v.^flds
}

sig Event {
  sender : Proc,           -- sender of event
  receiver : Proc,         -- receiver
  args : set Data,          -- arguments
  rets : set Data           -- return values
}{
  receiver != sender
  (args + reachable[args]) in sender.mayAccess.this
  (rets + reachable[rets]) in
    receiver.mayAccess.this + (args + reachable[args.flds])
}

sig Proc {
  owns : set Data,
  mayAccess : Data -> Event
}{
  all v : Data, e : Event |
    -- this proc may access value 'v' right before 'e' takes places only if
    v in mayAccess.e implies {
      -- (1) it owns 'v' or
      v in owns or
      -- (2) it has received 'v' as a parameter of a previous event 'e2'
      (some e2 : e.prevs |
        (this = e2.sender and v in e2.rets) or
        (this = e2.receiver and v in e2.args)) or
      -- (3) 'v' is a field of another data value that it has access to
      v in reachable[mayAccess.e]
    }
}

sig Private in Data {}
sig Untrusted in Proc {}{
  no owns & Private
}
```

Figure 5-5: Alloy encoding of our modeling framework.

- p already owns v (line 33),
- p has sent an event that yields v as a return parameter (line 36), or p has received an event that carries v as one of its arguments[10] (line 37), or
- v is reachable from one of the values that p can already access (line 39).

Let us now go back to the last two signature constraints for `Event` (lines 20-22). The first of these says that in order for a process to send an event, it must already have access to the event arguments (and all values that are reachable from it). The other constraint says that in order for a process to return a piece of data, it must already have access to that value (and all that are reachable from it). These constraints prevent the analysis from exploring spurious behaviors where a process is able to generate an event with a parameter that it does not have access to, which would lead to a trivial violation of a security property.

The last part of the model (lines 43-46) represents an encoding of the security concepts that we discussed in Section 4.3. The signature `Untrusted` is designated as a subset of `Proc` that behave maliciously[11], and a signature constraint is added to ensure that `Private` values are not initially accessible to `Untrusted` processes.

5.3.2 Translation

The translation from Poirot to Alloy is straightforward. The three basic constructs in a Poirot model—*components, data types,* and *operations*—are directly mapped to their counterpart Alloy signatures—`Proc`, `Data`, and `Event`. In addition, data and component fields are translated into fields inside their respective signatures, and operation guards are encoded as signature constraints[12].

Figure 5-6 shows a part of the Alloy model that has been translated from the Poirot model of the store system in Figure 5-2. The component `Store` and its associated fields are translated into an Alloy signature with the same name (lines 2-8). Note that each updatable field from Poirot is assigned an additional column of type `Event` in its Alloy counterpart. Since the language itself has no built-in notion of states, we employ a standard Alloy idiom in which the states of the system at different points in time are indexed by atoms of a designated type (in this case, `Event`). Consider the field `tokens`, which keeps track of the authentication tokens that have been assigned to users by a store (line 6). Given a particular store s and `Event` atom e, we can write the Alloy expression

$$s\texttt{.tokens.}e$$

to retrieve the part of the store state, as captured by `cart`, right before e is about to occur. For example, let us suppose that `cart` contains the following tuples, with event e_1 taking place immediately following e_0:

$$\{(s, u_{\texttt{alice}}, t_{\texttt{alice}}, e_0), (s, u_{\texttt{alice}}, t_{\texttt{alice}}, e_1), (s, u_{\texttt{eve}}, t_{\texttt{eve}}, e_1)\} = \texttt{tokens}$$

[10] `e.prevs` returns the set of all events that precede `e`, including `e` itself.

[11] The expression `sig A in B` introduces a subset relationship between `A` and `B`.

[12] This mirroring structure is not coincidental: The design of the Poirot DSL directly grew out of our experience in using Alloy as the primary modeling language during the early phase of this research project.

```
-- Store component, representing a set of processes
sig Store in Proc {
  catalog : ItemID lone -> lone ItemInfo,
  passwords : Username lone -> lone Password,
  -- updatable states are indexed on operations
  tokens : (Username lone -> lone Token) -> Event,
  carts : (Username -> ItemID) -> Event
}{
  all o : receiver.this |
    -- guards on events that this store receives
    Store_login[this, o] or Store_addItem[this, o] or
    Store_displayItem[this, o] Store_getItem[this, o]
  no sender.this     -- has no output events
  -- frame conditions on updatable states
  all o : Event - last | let o' = o.next |
    -- "tokens" may change only when this store receives a Login event
    tokens.o' != tokens.o implies (o in Login and o.receiver = this)
  all o : Event - last | let o' = o.next |
    -- "carts" may change only when this store receives a Login event
    carts.o' != carts.o implies (o in AddItem and o.receiver = this)
}
-- op Login[u: Username, p: Password, ret: Token] {
sig Login in Event {
  login_u : Username,
  login_p : Password,
  login_ret : Token
}{
  args = login_u + login_p and rets = login_ret
  sender in Customer and receiver in Store
}

pred Store_login[m : Store, o : Login] {
  let curr_tokens = m.tokens.o, next_tokens = m.tokens.(o.next) {
    -- ensures { passwords[u] == p }
    (m.passwords)[o.login_u] = o.login_p
    -- updates { make(t: Token) { tokens.insert(u ** t) and ret == t } }
    some t : Token | next_tokens = curr_tokens + o.login_u -> t and o.login_ret = t
  }
}

-- Customer component
sig Customer in Proc {
  id : Username,
  wishlist : set ItemID,
}{
  all o : sender.this |
    (Customer_login[this, o]) or (Customer_addItem[this, o]) or
    (Customer_displayItem[this,o]) or (Customer_getItem[this, o])

  no receiver.this     -- has no input events
}

pred Customer_addItem[m : Customer, o : AddItem] {
  -- calls { Store::AddItem.onlyIf {|o| o.(i).in?(wishlist)} }
  (o.addItem_i) in (m.wishlist)
}
```

Figure 5-6: Part of the Alloy encoding of the store model.

Retrieving the state of the store with e_0 and e_1 as indices gives us:

$$s.\texttt{tokens}.e_0 = \{(u_{\texttt{alice}}, t_{\texttt{alice}})\}$$
$$s.\texttt{tokens}.e_1 = \{(u_{\texttt{alice}}, t_{\texttt{alice}},), (u_{\texttt{eve}}, t_{\texttt{eve}})\}$$

Based on this, we can conclude that while performing e_0, the store has assigned a new token, $t_{\texttt{eve}}$, to the user identified by $u_{\texttt{eve}}$.

Each operation is also translated into a signature, with its argument and return parameters encoded as signature fields. Any guard condition or update statement associated with an operation is encapsulated inside an Alloy predicate. For example, the predicate `Store_login` evaluates to true if and only if (1) the given password matches that of the user to be logged in (line 35), and (2) a new token is assigned to the same user in the store's record and returned back to the sender of this request (line 37). Note that the constraint (1) is a precondition of the operation, whereas (2) is a statement about the world at the completion of the event (i.e., a postcondition). In our Alloy encoding, we do not make a distinction between pre- and postconditions, and require that each event satisfies their conjunction in order to be considered a valid event of the process.

A signature constraint is used to ensure that every event performed by a process is a valid one, by requiring the event to satisfy one of the operation predicates. For instance, the constraint for `Store` says that every input event that it performs must be one of the four types of store operation (lines 9-12). A similar constraint is used to define the set of events that a customer process is allowed to perform (lines 46-48); in this case, the customer may invoke any of the store services, but as discussed earlier, she would never request to add an item that does not belong to her wishlist (as specified in the predicate `Customer_addItem`

Another important part of a translated Alloy model is a set of *frame conditions*, which are used to ensure that the state of the system changes only when a process performs an event that directly operates on the state. Without frame conditions, an ensuing analysis may generate spurious scenarios where the state of the system arbitrary changes regardless of the events performed during the execution. During translation, a frame condition is generated for each updatable field, by determining all operations that may modify that field, and requiring that the content of that field change only as a result of performing one of those operations. For instance, the frame condition for `tokens` states that the assignment of tokens to users may only change when the latest event performed is a `Login` event (lines 15-17).

5.3.3 Encoding Representation Mappings

Alloy has a flexible type system that allows one to declare different sets of elements, and introduce an arbitrary relationship among those sets. For instance, consider the following fragment of Alloy:

```
sig E {}
sig A, B in E {}
```

This introduce a set of elements of type `E`, which itself contains two subsets `A` and `B`, but without specifying the relationship between `A` and `B`; it is left unspecified whether A intersects with B, or A and B together cover the entire set E. Consequently, every `E` element may belong to A, B, both, or neither of them. We may introduce an additional set in a separate part of the model:

```
sig C in E {}
```

If desired, we can use an Alloy fact to explicitly introduce a relationship between the subsets of E; for example,

```
fact { some A & C }
```

ensures that there is at least one element of E that belong to both A and C.

This feature of the type system, which allows an element to belong to multiple sets, and an existing type hierarchy to be extended freely, plays a crucial role in the implementation of our composition mechanism. Recall that, in our translation from Poirot to Alloy, every operation or data type is declared to be a subset of top-level Alloy signature Event or Data, respectively. Each of these subsets effectively assigns a representation to an event or a data value. For example, consider the following Alloy fragment:

```
sig Event {}
sig AddItem in Event {
  token : Token,
  item : ItemID
}
```

Suppose that some event atom $e \in$ Event also belongs to the set AddItem, and is associated with particular token t $\in$ Token and item i $\in$ ItemID. Conceptually, this particular combination of AddItem fields forms a possible description of e; i.e.,

$$e = \{\texttt{addItem}(\texttt{t},\texttt{i})\}$$

In a separate model, we may declare another type of event, to allow certain event atoms to be associated with multiple representations:

```
sig HttpReq in Event {
  url : URL,
  headers : set Header,
  body : Body,
  ret : RespMsg
}
```

When two models are brought together, we can use an Alloy fact to ensure that event e may belong to both AddItem and HttpReq if and only if the two groups of fields associated with these signatures satisfy some given constraint C. In other words,

$$e = \{\texttt{addItem}(\texttt{t},\texttt{i}), \texttt{httpReq}(\texttt{u},\texttt{hs},\texttt{b},\texttt{rm})\}$$

if and only if the following holds:

$$C(\texttt{t},\texttt{i},\texttt{u},\texttt{hs},\texttt{b},\texttt{rm})$$

This is the key idea behind the realization of our composition mechanism in Alloy.

Figure 5-7 shows how the mappings from the Poirot model of the store (Figure 5-4) are encoded in Alloy. The mapping constructs from the Poirot model are directly translated into Alloy predicates. The Alloy fact Mappings ensures that for each mapping, a pair of

```
1  fact Mappings {
2    all e : Event, s : Store, a : s.receives[AddItem],
3        r : Server, h : r.receives[HttpReq] |
4      -- (a, h) originate from the same process and
5      -- satisfy the mapping constraint
6      (s = r and mapping_addItemToReq[s,a,r,h]) iff
7        -- every event that is 'a' is also 'h' (and vice-versa)
8        (e = a iff e = h)
9
10   all d : Data, t : Token, c : Cookie |
11     -- (t, c) satisfiy the mapping constraint
12     mapping_tokenToCookie[t, c] iff
13       -- every data value that is 't' is also 'c', and vice-versa
14       (d = t iff d = c)
15 }
16
17 -- mapping between AddItem and HttpReq
18 pred mapping_addItemToReq[s: Store, a: AddItem, r: Server, h: HttpReq] {
19   a.addItem_t in h.httpReq_headers and
20   a.addItem_i in (h.httpReq_url).URL_query and
21   (h.httpReq_url).URL_host = r.Server_hostname and
22   (h.httpReq_url).URL_path = PathAddItem
23 }
24
25 -- mapping between Token and Cookie
26 pred mapping_tokenToHeader[t: Token, c: Cookie] {
27   c.Header_name = StrAuthToken and
28   c.Header_val = toStr[t]
29 }
```

Figure 5-7: Part of the Alloy encoding of the mappings between the store and HTTP models.

events or data value are considered to be the same element if and only if they together satisfy the predicate associated with the mapping. Note that an additional requirement is added to ensure that each pair of related events must be performed by the same process (`s = r` on line 6); without this, Alloy would allow a strange behavior where two events originating from physically distinct processes are considered to be the same.

5.4 Analysis

The Alloy Analyzer is a general-purpose constraint solver; meaning, each Alloy specification is essentially a set of constraints, and its analysis involves finding a satisfying instance to their conjunction. Checking a system model against a property, for example, can be formulated as finding an instance to a formula $S \wedge \neg P$, where S is a conjunction of constraints that describe the system behavior, and P is a statement of the property; if no instance is found, then one may conclude that the system satisfies the property (up to the analysis scope). The Alloy Analyzer, in turn, relies on a constraint solver called Kodkod [82], which translates a given first-order logic specification into an equisatisfiable CNF, and leverages an off-the-shelf SAT solver for instance generation.

The analysis in Alloy is exhaustive but bounded up to a user-specified scope on the size of the domains; in the context of Poirot, these bounds correspond to the number of processes, data values, and events as well as the length of event traces that will be analyzed. Although this implies that the analysis will not be able to discover a counterexample that lies outside these bounds, the user may re-run the analysis multiple times with larger scopes for increased confidence in the result.

Chapter 6

Case Studies

This chapter describes our experience applying Poirot to analyze the security of two publicly deployed systems: Handme.In and IFTTT. The goal of our studies was to answer the following questions:

- Can Poirot be used to find attacks that exploit the details of the system across multiple abstraction layers?
- Does our composition mechanism enable a modular reuse of domain models across multiple systems?
- Does Poirot's analysis scale to finding realistic attacks?

We will also discuss some of the technical challenges that we encountered, and the lessons learned from our experience in applying a model-based approach to the security analysis of realistic systems.

6.1 Methodology

During each case study, we carried out the following steps: (1) constructing a model of the system in Poirot, (2) performing an analysis in Alloy to generate potential attack scenarios and (3) confirming that those scenarios are indeed feasible on the actual system.

Model construction and analysis For each study, we took an incremental approach to building and analyzing the system model. We began by first constructing a high-level, abstract model of the system, describing its business workflow without details about its underlying platform. We then performed an initial analysis on this model to discover any potential violations of a given property; a counterexample at this level would correspond to a flaw in the design of the business or protocol logic.

Once the initial analysis was completed, we further elaborated parts of the original model by mapping them into relevant domain models from Poirot's library. Since both HandMe.In and IFTTT are web-based systems, some of the domain models used were common to both systems (e.g., models of an HTTP server and a browser), while others were unique to each of them (e.g., OAuth was relevant only for IFTTT). After each elaboration step, we re-analyzed the model against the same property, discovering attacks that were made possible by the newly introduced behavior of the system.

The process of constructing the model for each system will be discussed in more detail in Sections 6.3 and 6.4 [1].

Attack feasibility The output of an Alloy analysis is a counterexample trace demonstrating an attack on the *model* of the system. The events in this trace are still *approximations* of the real-world operations, and it is entirely possible that the depicted behavior is infeasible in the actual system, due to potential inaccuracies in the model.

In order to confirm that the attack described by a counterexample is actually feasible in reality, we manually converted the events into their concrete counterparts (i.e., HTTP requests), and replayed them on the actual system. To do this, we first created our own user accounts on the application site and populated them with sample data. To play the role of an attacker and interact with the site in a way that is not normally exposed to the user, we used web proxy tools such as Burp [71] and TamperData [44], constructing and sending our own hand-crafted HTTP requests. By observing the response from the application server, we were able to confirm whether the attack was indeed successful.

Our studies did not involve tampering with data in user accounts other than our own experimental ones.

6.2 Domain Models

Poirot contains a library of domain models that can be used to elaborate the user's input system model. The goal of the library is to achieve *reuse of knowledge*; building a domain model takes a considerable amount of effort and expertise, but **once** built by an expert, it should be reusable for analysis of **multiple** systems. For example, a web developer should not need to reconstruct a detailed model of a web browser and its vulnerabilities every time she wishes to analyze an application; this knowledge should already be available for systematic reuse.

Note that our use of the phrase *domain model* has a rather broad meaning; any model may qualify as long as it encodes a piece of knowledge that is generic and reusable across multiple systems. Types of descriptions in a domain model may include (but not limited to) protocols, architectural styles, features, algorithms, data structures, security vulnerabilities, and mitigations.

For our case studies, we were mainly interested in analyzing web applications, and so we constructed a number of domain models that describe different aspects of the web. We based models on reputable security sources such as OWASP [64] and CAPEC [59], and previous research efforts on formalizing the major components of the web [7, 29, 32]. Figure 6-1 shows the relationship between different domain models, which can be roughly grouped into the following categories:

[1]The complete models for the case studies, as well as the tool, are available at `http://people.csail.mit.edu/eskang/poirot`.

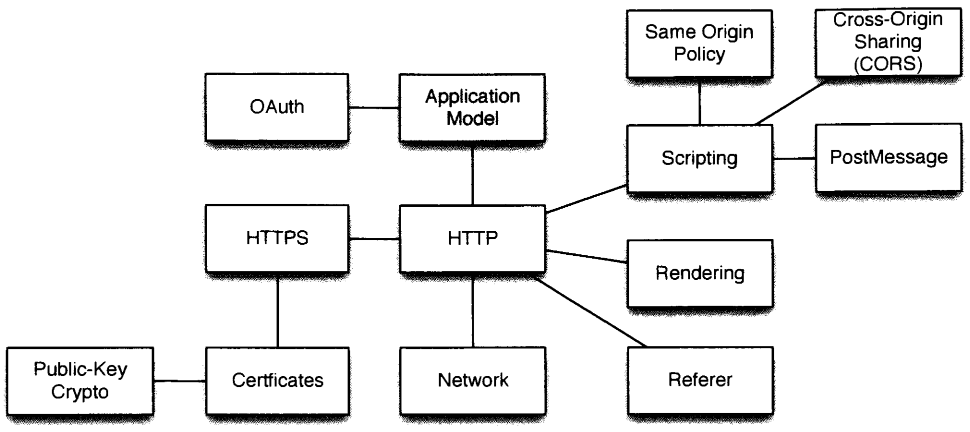


Figure 6-1: Web-related models in the Poirot library.

- Architecture models: Describe the major components of a communication layer and their interaction, including HTTP servers, clients, and network endpoints.
- Browser-related models: Describe the features of a standard browser, including in-browser scripts, policies to control their behavior (e.g., the same-origin policy), and rendering of web pages.
- Cryptographic models: Describe protocols used by components to securely communicate to each other, including SSL, certificates, and public-key encryption.

Of course, our library is far from being exhaustive, and may be missing domain knowledge that is crucial for discovering certain types of attacks. However, this is not an inherent limitations of our approach; in fact, we expect that Poirot's library to grow in size and applicability over time as more domain models are added. It is also worth noting that Poirot is not tied to a particular domain, and can be used to model other types of systems, as long as their behavior can be captured in our formalism.

6.3 HandMe.In

Handme.In (http://handme.in) is a web-based application designed to enable easy recovery of personal items. In a typical use case, an HandMe.In user purchases a sticker with a unique *code* written on it, and places the sticker on a physical item to be tracked. In an unfortunate situation where the item is misplaced, the person who finds item can notify its owner by entering the code and any relevant information, such as the location found and an arrangement to return the item in person, on the HandMe.In site. It currently has over 20,000 registered stickers.

A user can obtain a sticker either by being gifted it, or by purchasing it herself through the HandMe.In site. The process for the payment of stickers is delegated to Paypal, which offers a service called Instant Payment Notification (IPN). When a customer initiates a purchase, HandMe.In redirects her to the Paypal IPN site. After she makes a successful payment on Paypal, the IPN system will send the customer and billing information to the merchant site, which may then finalize the order.

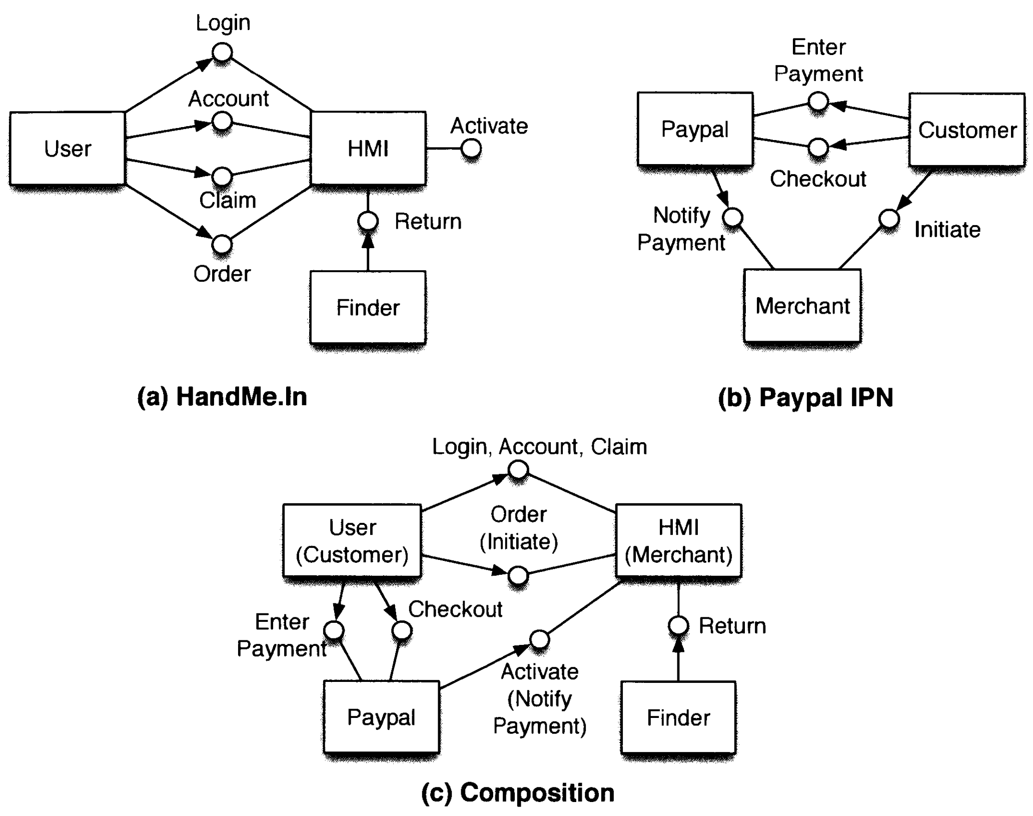


Figure 6-2: Graphical depictions of (a) HandMe.In, (b) the Paypal IPN protocol, and (c) their composition. Each box represents a component, a circle represents an operation, and a directed edge represents the invocation of an operation. Labels in the form of `A(B)` means that every instance of `A` can also be treated like a `B`. For simplicity, in (c), we will group three of the HandMe.In operations into one.

For this case study, we worked directly with the lead developer of HandMe.In to construct a model of the system. In addition, we consulted the online Paypal API documentation [69] to build a generic model of the IPN protocol. We were interested in analyzing two properties of the system: (1) Information about a lost item entered by the finder should be accessible only to the owner of the item, and (2) information given to the owner of a lost item must have come from the actual finder of the item.

6.3.1 Models

HandMe.In Figure 6-2(a) depicts a high-level design of the HandMe.In system, which involves interaction among three kinds of processes: the main HandMe.In application (`HMI`), a user who uses a sticker to track her item (`User`), and a person who discovers a lost item (`Finder`). The user and the finder interact with HMI by invoking various operations that are exported by the latter.

A part of the HandMe.In model in Poirot is shown in Figure 6-3; we will highlight some of its noteworthy features. The HandMe.In application keeps track of various types of information about its users in its database, such as the set of codes that have been activated for each user (field `activated` on line 16), and information about items that have been misplaced and subsequently found (`returned`, line 18). To model the physical process

```
# sticker codes, user passwords, and authentication tokens
privat data Code, Password, Token
# arrangement info for returning a lost item
privat data ReturnInfo
privat data AccountInfo[codes: (set Code), returns: (set ReturnInfo)]
data UserID
# The HandMe.In application
component HMI [
  # credentials of HMI users
  creds: UserID ** Password,
  # session IDs for users
  tokens: (updatable UserID ** Token),
  # codes that have been associated with users, but not activated
  linked: (updatable UserID ** Code),
  # codes that have been activated for users
  activated: (updatable UserID ** Code),
  # codes that have been returned
  returned: (updatable set Code ** ReturnInfo),
]{
  op Login[uid: UserID, pwd: Password, ret: Token] { ... }
  op Claim[code: Code, token: Token]{  ... }
  op Account[token: Token, ret: AccountInfo] { ... }

  op Order[token: Token, ret: Code] {
    ensures { no linked[tokens.token] } # no code already linked with the user
    updates {  # allocate a code and link it with the user identified by token
      uid = tokens.token
      make(c : Code) { linked.insert(uid ** c) and ret == c }
    }
  }
  op Activate[code: Code] {
    updates {  # activate the code that has been linked to the user
      uid = linked.code
      activated.insert(uid ** code) and linked.remove(uid ** code) }
  }
  op Return[code: Code, info: ReturnInfo]{
    # code can only be returned when it's already been activated for some user
    ensures { some activated.code }
    updates { returned.insert(code ** info) }       # mark code as returned
  }
}
# User of HMI
component User [
  uid: UserID, owns: (set Code), pwd: Password
]{
  invokes { HMI::Login }
  invokes { HMI::Claim }
  invokes { HMI::Account }
  invokes { HMI::Order}
}
# The person who returns a lost item
component Finder [
  code: Code, info: ReturnInfo
] {
  invokes { HMI::Return }
}
```

Figure 6-3: Part of a model of the HandMe.In system in Poirot

of discovering an item with a HandMe.In sticker on it, we will assign a code and return information to each `Finder` process, which may invoke the `Return` operation to notify `HMI` of the found item (lines 53 and 55).

When a user make an initial request to purchase a sticker, a new code is allocated for that user and recorded in the `linked` field (line 28). The code becomes activated and available for tracking only after the user completes the purchase of the sticker. The payment process is not handled by HandMe.In itself, and thus left unspecified in this model. Note that the `Activate` operation currently has no invoker, depicting a part of the system that has not been fully designed yet.

Paypal IPN The Paypal IPN, as illustrated in Figure 6-2(b), is a popular service that provides payment handling and notifications to third-party merchants. Figure 6-4 shows a part of the IPN model in Poirot. A typical workflow begins when a customer initiates a purchase with a merchant, which allocates an ID that will be used by both the merchant and Paypal to identify the item being purchased (line 12). The customer then takes this ID to Paypal for checkout, which initiates a new payment transaction (lines 28-31). Once the customer enters valid payment information (e.g., credit card number and billing address), Paypal will mark the transaction as having been completed (line 44).

After a successful transaction, Paypal notifies the merchant of the payment, along with other relevant data about the transaction (e.g., the item purchased, customer information, and the amount paid); in our model, this interaction between Paypal and a merchant is described by a guard condition that is attached to the invocation of `NotifyPayment` by `Paypal` (line 48). It is then up to the merchant to perform any further actions based on the transaction information; since `Merchant` is a generic description of merchants, this behavior is deliberately left unspecified (beside performing a check to ensure that the notification is received by the correct merchant, as on line 15).

Composition HandMe.In employs the Paypal IPN to handle the payment of its stickers. In order to reason about the impact of the IPN integration on security, we constructed a model that describes the combined behavior of the two systems, as shown in Figure 6-2(b).

A relationship between the two models mirrors how the IPN service is typically adapted by a merchant system. Intuitively, the `HMI` component can be regarded as playing the role of `Merchant`, with `User` acting like `Customer` from the Paypal model. In particular, the IPN service can be adapted so that when `HMI` is notified of a payment from `Paypal`, it immediately activates the code that corresponds to the item purchased during the transaction.

In our composition approach, this relationship is specified by mapping the `Order` operation to `Initiate`, and `Activate` to `NotifyPayment` in the Paypal model. Figure 6-5 shows the representation mappings used in the composition of the two models. The mapping constraint in `orderToInitiate` ensures that the code allocated for a newly ordered sticker correctly corresponds to the item that the user will be paying for during the Paypal checkout. Similarly, the constraint in `activateToInitiate` ensures that the code being activated is the one that has been paid for during the latest transaction.

Since both HandMe.In and Paypal are deployed as web systems, we wanted to reason about how the security of the system might be impacted by web-specific vulnerabilities. To do this, we took each abstract operation from the combined HMI-IPN model, and used

```
data ItemID, Amount, MerchantID, TxnID # transaction ID
data CustomerInfo  # information about customer (name, email, etc.)
# payment transaction info
privat data TxnInfo[mid: MerchantID, cinfo: CustomerInfo, item: ItemID, amtPaid: Amount]
privat data PaymentSecret  # e.g., credit card number

component Merchant [
  id: MerchantID
]{
  op Initiate[ret: ItemID] {
    # return the ID of the item to be purchased
    ensures { make(item: ItemID) { ret == item } }
  }
  op NotifyPayment[tinfo: TxnInfo] {
    ensures { tinfo.mid == id }
  }
}

component Paypal [
  ongoing: (updatable TxnID ** TxnInfo),     # ongoing transactions
  completed: (updatable TxnID ** TxnInfo),    # completed transactions
  validPayments: (set PaymentSecret)
]{
  op Checkout[item: ItemID, mid: MerchantID, ret: TxnID] {
    updates {
      # create a new transaction with the item ID
      make(tid: TxnID, txn: TxnInfo) {
        txn.item == item and txn.mid == mid and
        ongoing.insert(tid ** txn) and
        no ongoing[tid] and   # no duplicate transaction ID
        ret == tid                  # return the transaction ID
      }
    }
  }
  op EnterPayment[tid: TxnID, cinfo: CustomerInfo, amt: Amount, secret: PaymentSecret]
    # payment secret must be valid (i.e., valid credit card number), and
    # transaction must already exist
    ensures { secret.in? validPayments and some ongoing[tid] }
    updates {
      txn = ongoing[tid]
      # add customer and payment info to the transaction
      txn.cinfo == cinfo and txn.amtPaid = amt and
      # mark the transaction as complete
      completed.insert(tid ** txn)
    }
  }
  # notify the merchant of a transaction after it's been completed
  invokes { Merchant::NotifyPayment.onlyIf {|o| some completed.(o.tinfo) }}
}

component Customer [
  ci: CustomerInfo
]{
  invokes { Merchant::Initiate }
  invokes { Paypal::Checkout }
  invokes { Paypal::EnterPayment }
}
```

Figure 6-4: Part of the Paypal IPN model in Poirot

```
mapping orderToInitiate[h: HMI, o: Order, p: Paypal, i: Initiate] {
  # the code allocated for a new sticker is used as the ID of the item
  o.ret == i.ret
}

mapping activateToNotify[h: HMI, a: Activate, p: Paypal, n: NotifyPayment] {
  # the code activated matches the item in the transaction information
  a.code == n.tinfo.item
}
```

Figure 6-5: Selected mappings between the HandMe.In and Paypal IPN models.

```
property noReturnInfoLeak {
  # Information about a returned item should only be accessible
  # to the actual owner of the item.
  all(i: ReturnInfo, c: Code, h: HMI, m: Component){
    c.in?(m.owns) if mayAccess(m, i) and i.in?(h.returned[c])
  }
}

property noBadReturnInfo {
  # Only the actual finder of an item can enter the information
  # about the returned item.
  all(i: ReturnInfo, c: Code, h: HMI, m: Component){
    some(f: Finder) {
      i == f.info
    } if mayAccess(m, i) and i.in?(h.returned[c])
  }
}
```

Figure 6-6: Desired security properties of the HandMe.In system in Poirot

a representation mapping to describe its encoding as an HTTP request.

6.3.2 Security Issues

Figure 6-6 shows two properties that we analyzed against the HandMe.In system. The first of them, `noReturnInfoLeak`, is a confidentiality property, stating that return information about items should only be accessible to the user who owns the codes on those items. The second property, `noBadReturnInfo`, talks about the integrity of return information; that is, the system should not allow an arbitrary person to enter information about items that are owned by other users.

Our analysis generated a number of counterexamples, two of which we confirmed to be feasible attacks on the system; they were both previously unknown to the developer of the system:

Guessable sticker codes Although the codes on stickers are unique, they are arranged in a simple, incremental ordering (`147`, `148`, ...), and so are easily guessable. This allows the attacker to violate the property `noBadReturnInfo` by entering a randomly guessed code and bogus information about the item recovery. While this might seem like an innocuous attack, it can be easily carried out on a large number of codes, potentially causing inconvenience to many users on the site. This attack involves details only at the business logic layer, and was generated by Poirot on our initial design model.

Missing check during payment The IPN service provides no guarantee that the person paying for a product is the same person who will receive the product. Paypal sends back information about the payer (e-mail, billing address, etc.,), and the merchant may perform further checks to ensure that the product will be delivered to the same person. HandMe.In does not perform such checks, because it assumes that a user will be directed to the IPN site only by following the standard workflow on HandMe.In. This assumption is reasonable at the business logic layer, but is violated when the user is interacting with the site through a browser. In particular, the site has a cross-site scripting (XSS) vulnerability on one of its pages, allowing an attacker to insert a link that appears legitimate, but which, in fact, redirects the victim user to the IPN for a sticker that has already been assigned to the attacker. As a result, the victim may willingly pay for the sticker that will be owned by the attacker, leading to a violation of the property `noReturnInfoLeak`. This attack combines details at both the business logic and HTTP protocol layers, and was generated by Poirot after the initial model was elaborated with the HTTP domain model.

We notified the result of our analysis to the developer; all of the above security issues have since been addressed.

6.4 IFTTT

IFTTT (short for "If-This-Then-That") is a web-based system that allows a user to connect and automate tasks from independent web services using simple conditional statements. The basic building block of IFTTT is a *channel*, a service that exports a number of functions through its API. IFTTT allows a user to construct a *recipe*, which consists of two channel functions: a *trigger* and an *action*. Once the recipe is registered, each time a trigger is performed, IFTTT automatically executes the corresponding action. For example, a recipe can be made so that whenever the user is tagged in a photo on Facebook, IFTTT creates a post containing the photo and its caption on the user's Google Blogger account.

Since IFTTT performs tasks automatically on the user's behalf, potentially accessing private data in the process, the user must explicitly authorize IFTTT to do so through the selected channels. For example, before registering the above sample recipe, the user must give IFTTT (1) a permission to access photos on Facebook, and (2) a permission to create new blog posts on Google Blogger.

The goal of this case study was to analyze whether IFTTT channels could be composed in an insecure way, allowing a malicious person to access information that would not have been possible without IFTTT. In particular, we analyzed the following security property: A user's private data from one channel should only be accessible to the same user's accounts on other channels.

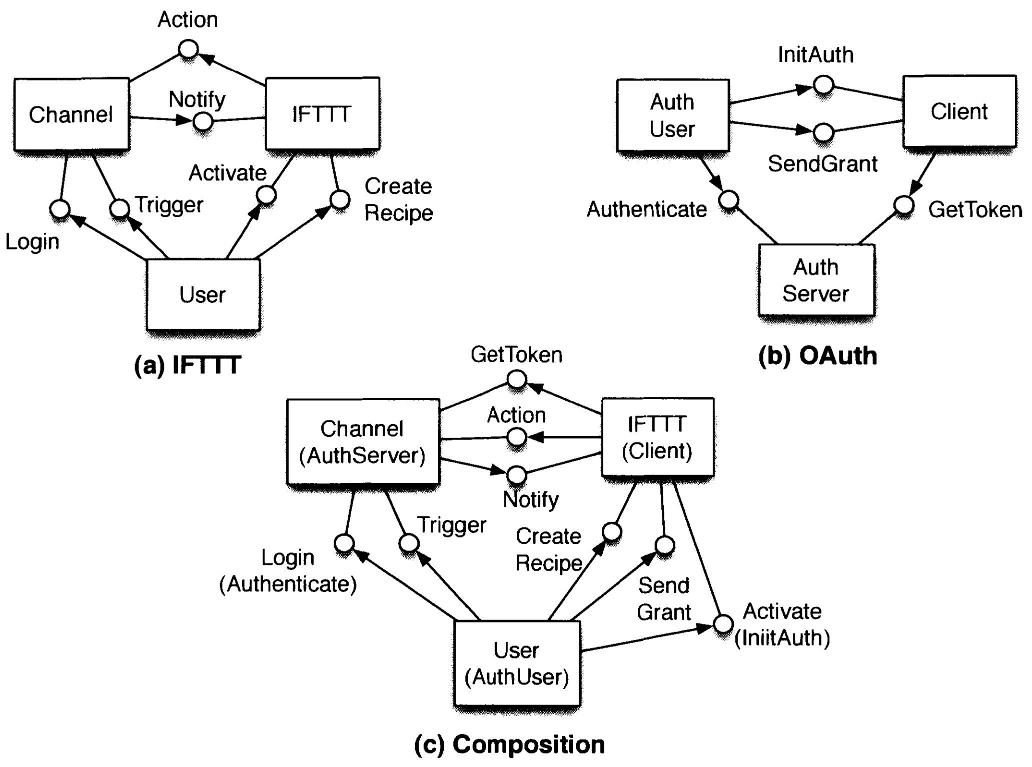


Figure 6-7: Graphical depictions of (a) IFTTT, (b) the OAuth protocol, and (c) their composition.

6.4.1 Models

Figure 6-7(a) shows a high-level design of the IFTTT system, which involves interaction among three different types of processes: the IFTTT application, channels, and users who wish to automate tasks between those channels. Since our goal was to analyze the security of IFTTT's service composition mechanism—instead of looking for flaws in a particular web service—we did not explicitly model the details of web services themselves. Instead, we built archetypal descriptions of channels, triggers, and actions that over-approximate all possible dataflow throughout the system.

IFTTT application A part of the IFTTT model in Poirot is shown in Figure 6-8. One of the key concepts in this system is the notion of a *recipe*, here represented as a data type that consists of the IDs of a trigger and its corresponding action (line 10). The IFTTT application keeps track of a list of currently registered recipes for each user (line 17). If a user wishes to register a new recipe under her account, she may invoke the operation `CreateRecipe` and provide the recipe along with her credentials.

Before IFTTT can perform a recipe, it must be able to access the services that are offered by both the trigger and action channels. In the IFTTT terminology, the user must *activate* a particular channel and grant IFTTT an *authorization token* that can be used to access that channel. The process of obtaining a token varies between channels, and thus, is left unspecified in this model (line 29); later, we will describe how this part of the design can be elaborated with a model of OAuth, a popular third-party authorization protocol and

```
data UserID, ChannelID
# identifiers for triggers and actions
data TriggerID, ActionID
privat data Password  #  user crendential
# token used by IFTTT to perform actions on channels
privat data Token
# data passed in as part of payload
privat data Payload
# IFTTT recipe, consisting of a trigger and an action
data Recipe[tid: TriggerID, aid: ActionID]

# IFTTT server
component IFTTT [
  # user credentials
  pwds: UserID ** Password,
  # currently stored recipes
  recipes: (updatable UserID ** Recipe),
  # tokens associated with channels
  tokens: (updatable ChannelID ** Token)
]{
  op CreateRecipe[uid: UserID, r: Recipe, c: Password] {
    # can only create a recipe if the right credential provided
    ensures { c == pwds[uid] }
    # create a new recipe
    updates { recipes.insert(uid ** r) }
  }

  # obtain an access token for a new channel
  op Activate[uid: UserID, cid: ChannelID] { ... } # unspecified

  # poll a channel for a trigger, and perform an action only if
  invokes { Channel::Poll.then Channel::Action do |p, a|
      # if there's some recipe such that
      some(r: Recipe) {
        r == recipes[a.uid] and
        # the polled trigger is linked to the action and
        r.tid == p.tid and r.aid == a.aid and
        # the payload from the trigger is provided as input to the action
        a.uid == p.uid and a.p == p.ret
      }
    end
  }
}
```

Figure 6-8: Part of the IFTTT model describing the main application and recipes.

the most common method of obtaining a token among IFTTT channels. Once it successfully obtains the token, IFTTT will store it in its record (`tokens` on line 19), to be retrieved whenever a recipe with the corresponding channel is triggered.

Later, we will return to this part of the model and describe how IFTTT performs a recipe (lines 32-39).

Channel Figure 6-9 shows the part of the IFTTT model that describes channels and users. Each channel is associated with a channel ID, and itself keeps track of a list of user passwords as well as tokens that can be used to access its services (lines 3-7). To model the behavior of a *generic* channel, we introduced two abstract operations called `Action` and `Trigger`. Each action is associated with an ID, and can be performed only when provided with a token that correctly identifies the user (line 28). Also, the action is associated with a *payload*, which represents generic data that are passed between the channel and its users.

Every time an action is performed, its associated payload is recorded in the channel history (line 15). A user may then look up the actions that have been performed on her account, and access any of the related payloads (operation `Lookup` on line 18). This is done as a way of allowing dataflow from one user to another through a pair of trigger-action events; we will come back to this point shortly.

The structure of the `Trigger` operation is similar to that of `Action`; it is associated with an ID, and can only be performed when its invoker provides a correct token (line 28). Like with actions, each trigger is associated with a payload, and its occurrence is recorded by the channel in field `triggerBuffer` (line 11).

Performing a recipe In order to perform a registered recipe, the IFTTT application periodically *polls* the trigger channel and inquires whether the trigger has been recently executed. If so, the `Poll` operation returns the associated payload back to IFTTT (line 38), and removes the record of that trigger from the buffer (line 39).

Let us recall the model of the IFTTT application in Figure 6-8. In Poirot, the expression

```
invokes{X.then Y do |x, y| C(x,y) end}
```

restricts the behavior of a process so that it is allowed to perform event `x` ∈ `X` only if (1) its previous event was `y` ∈ `Y` and (2) `x` and `y` satisfy some constraint `C`.

This specification idiom is used to describe how the IFTTT application performs a recipe in response to the occurrence of a trigger (lines 32-39). In particular, IFTTT may invoke some particular action (a) only after it successfully polls the corresponding trigger channel; in addition, it directly passes the payload returned from the poll to the action (`a.p == p.ret` on line 39).

Composition with OAuth OAuth 2.0, illustrated in Figure 6-7(b), is a protocol used to carry out for third-party authorization [40]; that is, it allows an application (called `Client`) to access resources from another service provider (`AuthServer`), pending the approval of the user who owns those resources (`AuthUser`). A typical workflow begins when the user attempts to perform an operation on the client application that requires authorization from `AuthServer` (`InitAuth` operation). The user is first required to prove her identity with

```
component Channel [
  # channel ID
  id: ChannelID,
  # credentials for channel users
  pwds: UserID ** Password,
  # tokens used to access this channel
  tokens: UserID ** Token,
  # triggers
  trigs: (set TriggerID),
  # used to keep track of triggers that have been performed
  triggerBuffer: (updatable (TriggerID ** UserID) ** Payload),
  # actions
  actions: (set ActionID),
  # used to keep track of actions performed so far
  actionHistory: (updatable (ActionID ** UserID) ** Payload),
]{
  op Login[uid: UserID, pwd: Password, ret: Token] { ... }
  op Lookup[aid: ActionID, uid: UserID, t: Token, ret: Payload] { ... }

  op Action[aid: ActionID, uid: UserID, t: Token, p: Payload] {
    # can perform action only if the right credential or token presented
    ensures { aid.in?(actions) and t == tokens[uid] }
    # mark this action as being performed
    updates { actionHistory.insert((aid ** uid) ** p) }
  }

  op Trigger[tid: TriggerID, uid: UserID, t: Token, p: Payload] {
    # can perform trigger only when the right user credential presented
    ensures { tid.in?(trigs) and t == tokens[uid] }
    # mark this trigger as being performed
    updates { triggerBuffer.insert((tid ** uid) ** p) }
  }

  op Poll[tid: TriggerID, uid: UserID, t: Token, ret: Payload]{
    # check to see whether trigger has been performed
    # if so, return the payload associated with that trigger
    ensures { t == tokens[uid] and
       some (p: Payload) { p == triggerBuffer[tid][uid] and ret == p } }
    updates { triggerBuffer.remove((tid ** uid) ** p) }
  }
}

component User [
  id: UserID,  pwd: Password, owns: (set Payload)
]{
  invokes { IFTTT::CreateRecipe }
  invokes { iFTTT::Activate }
  invokes { Channel::Login }
  invokes { Channel::Trigger }
}

property noPrivateInfoLeak {
  # Payload can only be accessed by users who own it
  all(p: Payload, u: User) {
    p.in?(u.owns) if mayAccess(u, p)
  }
}
```

Figure 6-9: Part of the IFTTT model describing channels and users.

AuthServer, typically by logging onto the site (Authenticate). Once it identifies the user, AuthServer will return an *authorization grant* back to the user, who then forwards it to Client (SendGrant). In the final step, the client exchanges the grant for an *authorization token*, which can subsequently be used to access resources on AuthServer (GetToken).

OAuth is one of the most common methods for granting IFTTT access to channel services. To reason about the behavior of IFTTT in conjunction with OAuth, we performed a composition of the two models, resulting in an elaborated model as shown in Figure 6-7(c). The relationship between the two reflects how OAuth is adapted by IFTTT in practice: IFTTT plays the role of Client, with channels serving as AuthServer and IFTTT users as AuthUser. In the context of IFTTT, the OAuth workflow begins when the user attempts to activate a particular channel; to express this relationship, we introduced a representation mapping between Activate and InitAuth from the two models. In addition, the task of AuthUser proving her identity to AuthServer can be accomplished by the user logging onto the corresponding channel (Login in the IFTTT model); in our composition, an additional mapping was used to link the two operations.

6.4.2 Security Issues

Through the use of recipes, IFTTT provides a simple, easy way for users to connect and automate tasks of multiple channels. But this convenience is not without its potential downfalls; since IFTTT enables flow of data between channels that normally do not communicate with each other, there is an added risk of unintended data exposure.

For our study, we were interested in checking whether a malicious actor could exploit the IFTTT system to access information belonging to another user on a channel. In our model, this is specified as a simple confidentiality property that any payload originating from a user should be accessible only to that same user throughout the system execution (lines 54-55 in Figure 6-9). During our analysis, we discovered two feasible attacks on the IFTTT system; both attacks exploit details across the IFTTT and HTTP protocol layers, and were detected by Poirot after the initial model was elaborated with the HTTP model:

Privilege escalation A pairing of a trigger and an action that require different levels of privilege can be exploited by an attacker to gain unintended access to the action channel. For example, Blogger is designed with an assumption that only those with the right credential can create a new post on a user's account, and so it treats all input post data as having come from a trusted source. On Facebook, however, a user may unknowingly be tagged on a photo that is owned by another (potentially malicious) user; this means that the trigger can be performed at the attacker's will, leading to a new post on the victim user's blog with the attacker's photo and caption. This gives the attacker an ability to indirectly manipulate the victim's blog through IFTTT—for example, by encoding malicious data (e.g., XSS code) into the caption. This form of privilege escalation in IFTTT is a known issue that has been reported by security researchers [26].

Information leakage with login CSRF Login CSRF is a type of a browser attack that manipulates a victim into logging onto a site under the attacker's credential. Normally, login CSRF is considered a relatively minor form of attack, because at most it allows the attacker

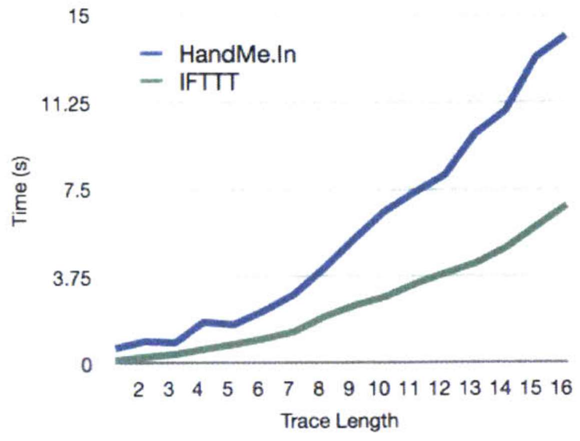


Figure 6-10: Average analysis times over trace lengths.

to access a log of the victim's actions (e.g., search history on Google). In combination with IFTTT, however, it can be used to carry out an attack with far more serious consequences.

Consider a situation in which a victim user is unknowingly logged onto a site that corresponds to one of the channels available on IFTTT. The user proceeds to create a new recipe with an action on that same channel; since the user is already logged onto the site (albeit as the attacker), she will not be prompted to enter her credential again. Consequently, the action channel on the registered recipe will be associated with the attacker's account. This means that, for example, any data from the recipe's trigger will flow directly into the attacker's account on the action channel.

As far as we are aware, this is a previously unknown issue with IFTTT. To demonstrate the feasibility of the attack, we selected 15 channels with an action that can be used to store information from a trigger, and inspected their login page to see whether they have a built-in protection against CSRF. We found that 4 out of those sites lack the protection, and notified them of the potential security issue; 3 of them have since addressed the issue.

6.5 Analysis Performance

We evaluate the scalability of Poirot's analysis over the two case studies. As discussed in Section 5.4, the analysis relies on constraint solving over finite domains, and so it must be given an explicit scope to bound the number of processes, data values, events, and the maximum length of traces. Figure 6-10 shows the average analysis times for the two cases studies as the maximum length of traces is varied; we used a fixed scope of 8 for processes and 12 for data values. All analyses were performed on a Mac OS X 1.8 GHz Core machine with 4G RAM; Lingeling [17] was the SAT solver used.

Figure 6-10 shows an exponential growth trend for the analysis times as the trace length increases; this is not surprising, since the number of possible event combinations that must be explored also grows exponentially. The shortest trace corresponding to a counterexample had 4 events, and the longest trace had 11 events; we performed additional analysis up to the trace length of 16 without discovering any more counterexamples. In all cases, the analysis took under 15 seconds. The results do not necessarily imply that the checked properties are valid, as there might exist a counterexample beyond the maximum scope that we used. However, based on our experience applying Poirot to numerous examples, we believe that the bounds we used were large enough to capture many common web security attacks.

6.6 Challenges and Lessons Learned

We have used Poirot to model and analyze a number of small and large systems, the most complex ones being the two systems above. We were successfully able to reuse the domain models across most of these systems, in part because we invested considerable effort (6 months) into ensuring the generality of the models; based on our experience, we believe that reusability justifies this upfront cost. We also expect Poirot's library to grow in size and applicability over time. For example, we initially created a model of the OAuth protocol [40] in order to analyze it, and we were able to reuse the same model as part of the analysis of IFTTT.

One of the most challenging parts of the case studies was building faithful models of the systems. As with many commercial systems, the source code was not available, and so we had to leverage other available means to gain an understanding of the systems. For Handme.In, we directly consulted the main developer of the site for the information about its web API and its integration with Paypal. Most of the triggers and actions offered by IFTTT channels have an online API documentation available; for those without reliable documentation, we manually generated and inspected the HTTP requests using a web proxy tool (in particular, Burp [71]) and approximate the behavior of the system.

We found the ability to partially specify representation mappings especially helpful in two ways. First, the ability to automatically explore attacker capabilities (as described in Section 4.3.1) was crucial in discovering subtle, complex attacks; it would have been challenging to do so using an approach that requires each attack model to be manually instantiated against the system model (e.g., [30]). Second, being able explore a space of possible design configurations was particularly useful for the IFTTT study, which involved analyzing the system against *any* potential set of web services, instead of a particular one.

Chapter 7

Related Work

7.1 Composition

Our work was strongly influenced by previous research on *views* in software engineering [28, 41, 62, 73]. In a typical development process, various stakeholders may have differing views on the system, hampering the construction of a single, coherent global model of the system. In the context of security, the attacker can be regarded as one type of stakeholders (with a malicious intent to sabotage the system), exploiting details in a view that differs from that of the designer. Most of these efforts were mainly concerned with achieving *consistency* in the global model; that is, dealing with views that may contain conflicting, contradictory statements about the system.

Model merging is an active line of research on techniques and tools for composing distinct but possibly overlapping models. Merging techniques have been developed for various types of models, including architectural views [53], behavioral models [15, 61, 83], database schemas [72], and requirement specifications [76]. Among these, the works on behavioral models are most closely related to our work [15, 61, 83]. A common property achieved by the existing merging frameworks is the preservation of behavior: that is, when two models M_1 and M_2 are merged, the resulting model M' refines the behavior of both M_1 and M_2. In comparison, our composition operator does not aim to provide such behavioral guarantee; in general, M' may not be a refinement of M_1 nor M_2. Instead, our goal is to capture different ways in which a property in original model M_1 may be violated by added behavior from M_2.

Another work closely related to our approach is Georg and her colleagues' work on an aspect-oriented approach to security modeling and analysis [30]. In this approach, a set of generic attack models (called *security aspects*) are instantiated against a primary system model, and the Alloy Analyzer is used to check the composed model against a security property. Our approach differs from theirs in two ways. First, during the instantiation step, the user must provide a full correspondence between two models, unlike our approach where a partial mapping is sufficient for performing an analysis. In addition, our notion of representation is more general than their notion of correspondence, which is limited to a mapping between the *names* of modeling elements. Their approach cannot, for example, express a more complex mapping that relates the *structures* of two elements (e.g., encoding `addItem` as `req`).

7.2 Abstractions and Security

In his seminal paper [49], Lampson describes the problem of restricting data to a single program, and discusses how an attacker may achieve leakage through various methods that are unanticipated by the programmer; these methods are referred as *convert channels*. This paper was a major source of inspiration for our work—in particular, the idea that a piece of information may be conveyed in different representations, only some of which may be known to the programmer.

Full abstraction is a notion that was originally introduced to reason about equivalence between the operational and denotational semantics of a programming language [57, 70]. This notion has since been applied in the context of security, to reason about potential security issues when a description of a system in one language is translated to another [1]. Informally, a translation from language L_1 to L_2 is considered *fully abstract* if, for any given system description S in L_1 and its counterpart S' in L_2, a malicious agent is not able to extract more information by observing S' than it can from S. For instance, the translation mechanism from C# to the .NET Intermediate Language (IL) in an earlier version of the compiler framework was not fully abstract, allowing an attacker to manipulate an IL program in a way that was not possible at the C# layer [46].

Our work can be regarded as a systematic approach to checking whether an encoding scheme from one model of a system to another—corresponding to a set of representation mappings in our framework—satisfies a *weaker* form of full abstraction; that is, whether it preserves a *particular* property that has been established in the original model of the system. We are interested in further exploring the relationship between our work and full abstraction, in particular extending our analysis technique to help designers construct a translation scheme between abstractions that ensure a certain type of full abstraction.

The term *representation* has a long history in computer science. One of the most powerful concepts in programming language design is the notion of an *abstract data type* (ADT), which is used to decouple the abstract interface of a data structure from its underlying representations [50, 51]. For example, a `Set` data type in a program may be implemented by encoding the set as a list of elements, but this detail is kept private from any client of the `Set` interface. As a result, the programmer will be able to substitute a different implementation of `Set`, while maintaining the properties of the interface the client relies on (also called *representation invariants*).

Our use of the term *representation* is similar, with one major difference. Most programming languages that support ADTs have a static or runtime mechanism to guarantee the separation between an ADT and its representation. For example, in Java, the programmer may use the keyword `private` to protect certain members of a class; any external attempt to access these members will be detected and denied during the compilation step. In comparison, we make a modeling assumption that in general, no such protection mechanism exists for most systems, and the attacker may have access to all possible representations of a system entity. By manipulating the details of a representation, the attacker may be able to undermine a security invariant that was established on a previous model of the system.

7.3 Protocol Modeling and Analysis

A large body of work exists on formally verifying models of systems and protocols for security. Most protocol languages describe a system in terms of abstract agents and messages between them, and are not designed for elaborating their underlying representations [2, 52, 81]. Several protocols that were proven to be secure have suffered attacks when deployed on real systems [10], because the models used in the proofs omitted details that were exploitable by an attacker. Recent works in program verification use semi-automated theorem provers such as Coq [16] and Boogie [13] to verify properties across multiple layers of a system [21, 36, 47]. While these approaches can provide strong end-to-end guarantees, constructing a proof requires a significant amount of human effort, and may not be cost-effective in early design stages, where alternative decisions are still being explored.

Codifying domain knowledge for reuse is not a new idea, and has also been applied in the context of security analysis. Bansal et al. constructed a library of web-related models (called WebSpi) in the ProVerif language [18], and used it to analyze the security of several websites [12]; however, they do not provide a general composition mechanism, and the library exists as a monolithic model that is not easily extensible. Almorsy et al. created common security attack patterns in the Object Constraint Language (OCL), and used them to identify architectural risks [9]. Our approach is complementary to theirs, in that the two produce different types of feedback to the designer; their approach is capable of producing various security metrics (such as the size of attack surface), whereas our focus is on performing an exhaustive behavior analysis of a model against a security property.

7.4 Refinement

A traditional approach to rigorous software development is based on the notion of *refinement* [38, 88]. Here, the designer starts with an abstract design of the system and incrementally refines it into a more concrete one, at each step ensuring the conformance of the concrete system against its abstract predecessor.

In practice, this approach may not always be feasible. Given the complexity of modern systems, the designer rarely has the luxury of building a system from scratch; instead, starting from a high-level design, he reuses existing components to implement various parts of the design, inadvertently introducing behavior that may invalidate a property of the design. Furthermore, the designer may not have the freedom to modify the existing components, forcing him to return to the abstract design to address the issue that originates from a lower layer; current browsers will remain vulnerable to attacks such as CSRF, and so the designer must deal with them by strengthening the authentication logic in the higher-level design.

Event-B [5] is a specification and analysis framework designed to aid the construction of a correct system through a series of refinement steps. Event-B provides various ways for refining an abstract machine by, for example, replacing an abstract event with a concrete one, splitting an event into multiple concrete events, introducing a new event, or merging existing ones. The refinement patterns in Event-B are, in some sense, more elaborate than our mapping, which currently does not allow an event from one process to be refined *hierarchically* by a group of events from another process.

On the other hand, our mechanism offers greater flexibility in that it allows (1) a system to be partially refined, and (2) an entity to be associated with multiple representations *simultaneously*. In comparison, Event-B requires every event in an abstract machine to be refined during every step of refinement; so, this approach could not be used, for example, to elaborate our model of the online store with OAuth, where only a part of the store is mapped to the protocol model. Similarly, in Event-B, once an event is refined into a concrete one, the alphabet term representing the abstract event is no longer available in the resulting mode. Consequently, this approach would not allow the store model to be elaborated with the HTTP and OAuth models in an arbitrary order; after the first step, the abstract events from the store model would no longer be available for further composition.

Chapter 8

Discussion

In this chapter, we will discuss limitations of our approach, and propose possible directions for improving its applicability and effectiveness. We will then conclude this thesis with a reflection on our experience during this research project.

8.1 Limitations

Hierarchical mapping In our modeling approach, all events are treated as being atomic, instantaneous actions. But sometimes it may be desirable to specify a certain event as itself consisting of a set of more detailed events performed in a particular order. For instance, an HTTP request, represented as a single event at a high-level of abstraction, may actually involve a series of handshakes between the server and the client. To accurately model such *hierarchical* relationships, our mapping could be extended to allow an event to be mapped to a *sequence* of events.

For instance, suppose that a representation mapping for events is now a set of tuples belonging to $R_E \times Seq(R_E)$; then, we could specify a mapping that describes how a typical HTTP request is implemented as a sequence of low-level network requests [80]:

$$(\texttt{req}, \langle \texttt{lookupDNS}, \texttt{tcpConnect}, \texttt{tcpSend}, \texttt{tcpWait}, \texttt{tcpLoad}, \texttt{tcpClose} \rangle)$$

But this extension is likely to involve non-trivial modifications to our composition mechanism, as it would no longer be sufficient for a pair of processes to synchronize on single events. Instead, we would need to introduce a more complex type of synchronization that aligns two event sequences of different lengths. Prior works on relating event structures, such as *action refinement* [84], may provide a good starting point for exploring such an extension.

Modeling probabilistic behavior Our modeling approach is based entirely on a discrete logic, and sometimes insufficient for capturing certain types of behavior in the real world—especially those involving human agents. Since our processes behave in a non-deterministic manner (to the extent allowed by their guards), the analysis will deliberately choose the actions that lead to a violation of a given property. This means, for example, that a human process will behave like a clueless user, always falling for phishing attacks or blindly browsing to a malicious web page. But this is a rather extreme view of user behavior; provided with a clear, well-designed set of security warnings, users are capable of making an informed decision while browsing the web [8].

A similar issue exists in our model of dataflow. Currently, a process is allowed access to a piece of data only if it already owns the data or receives it through an event. This means, by definition, that the process is not able to obtain new information through guessing or statistical inference—again, a rather limited view of how an attacker interacts with systems in the real world.

In order to capture a more realistic model of the above two, we would need to explore other types of behavioral models beside ones that are purely logic-based (like ours). A couple of approaches seem promising in this direction, including efforts to reconcile formal and complexity-based models of cryptography [4], and works on probabilistic models of information flow [37].

Other security properties Poirot currently allows the user to specify and check *trace properties*—a kind of property that can be evaluated by inspecting a single execution trace (e.g., "nothing bad ever happens"). However, certain classes of security properties inherently talk about multiple traces of a system; these are also called *hyperproperties* [23]. For instance, a *non-interference* property says that an attacker should not be able to learn new information by observing how the system behavior changes when other users participate in its services. In order to analyze such properties, our analysis technique would need to be extended to perform a higher-order reasoning, where *sets* of traces are explored at a time (instead of individual traces) to detect a potential violation of a property.

8.2 Future Directions

8.2.1 Secure Mapping Synthesis

In Section 4.2, we introduced the problem of *mapping generation*, which involves, given a partial specification of a representation mapping, generating a full set of entries in the mapping that leads to a violation of a given property. Recall that the problem can be formulated as finding a set of witnesses to the following formula:

$$\begin{aligned} &\mathop{\exists}_{i=1}^{n-1} M_i \in M \bullet \Big(\bigwedge_{j=1}^{n-1} sat(M_j, S_j)\Big) \wedge \\ &\quad \exists\, Sys \in P \bullet Sys = compose(\{p_1, p_2, ..., p_n\}, \{M_1, M_2, ...M_{n-1}\}) \wedge \\ &\qquad \exists\, t \in traces(Sys) \bullet \neg Prop(t) \end{aligned}$$

for n processes $(p_1, p_2, ..., p_n)$ and $n-1$ mapping specifications $(S_1, S_2, ..., S_{n-1})$.

Instead of generating an insecure mapping, we may be able to synthesize a *secure mapping*, which, when used for composition, ensures that the resulting system satisfies a given property. This problem can be stated by slightly modifying the last part of the above formula, as follows:

$$\begin{aligned}&\mathop{\exists}_{i=1}^{n-1} M_i \in M \bullet (\bigwedge_{j=1}^{n-1} sat(M_j, S_j)) \wedge \\ &\quad \exists\, Sys \in P \bullet Sys = compose(\{p_1, p_2, ..., p_n\}, \{M_1, M_2, ...M_{n-1}\}) \wedge \\ &\qquad \forall\, t \in traces(Sys) \bullet Prop(t)\end{aligned}$$

That is, the problem now involves finding a set of witness mappings such that in every trace of the resulting system, *Sys*, the property *Prop* holds.

Successfully implementing this synthesis framework will involve at least two challenges. First, the problem requires a different kind of analysis than the one that is currently performed by Poirot, as it involves a higher-order reasoning over all possible traces of a system; a recent extension to the Alloy Analyzer that allows reasoning over higher-order quantifiers may provide an initial solution [56].

Another challenge is synthesizing a secure mapping that the user will find suitable for an actual implementation. In principle, a secure mapping always exists: An empty mapping, which maintains a complete separation between two given processes, will preserve all properties that have been previously established in those processes. But this mapping is also not a very useful one! In practice, the user is likely to provide some initial partial specification of a mapping, which would then be completed by the synthesis engine. However, it is possible that no completion would ensure a given property (for example, when a high-level program is embedded into a platform that allows an attacker to trivially take control of the program). The challenge in this case will be to provide useful feedback to the user, explaining why no secure mapping exists, and suggesting possible ways to modify the original processes to allow such mappings.

8.2.2 Alternative Analysis Approaches

The analysis performed by Poirot (which, in turn, relies on the Alloy Analyzer) is bounded; that is, even if the analyzer fails to find a violation of a given property, there may still exist a counterexample that involves a larger number of processes, events in a trace, or data values beyond the scope given by the user. As an alternative backend to Alloy, we could instead translate Poirot models into the input language of an SMT solver [14], which provides stronger theoretical guarantees for certain decidable theories (such as strings and integer arithmetics). This alternative translation, however, would involve restricting the input language of Poirot—especially with respect to quantifiers, which often cannot be discharged by an SMT solver without manual guidance from the user.

Another interesting question is whether there exists a conservative, general upper bound (on the number of processes or trace lengths) for the type of models and properties that can be analyzed in Poirot. Such a bound, if it exists, would allow us to perform an analysis on a finite-domain system, and still conclude that the property holds for systems of all sizes beyond that bound. While coming up with such a bound is a non-trivial task, there have been some successful efforts in the past for restricted classes of security protocols [22, 24]. Based on these works as starting points, we plan to investigate whether we can identify a subset of Poirot that yields a general upper bound, but is still expressive enough for modeling a wide variety of systems.

8.2.3 Other Applications

In this thesis, we have focused mostly on studying the security of web-based systems. However, we believe that our approach is applicable to systems from other domains, including, but not limited to:

- **Safety-critical systems**: Systems such as medical devices, automotive systems, and public infrastructures are increasingly being controlled by software, allowing access from external actors even without physical presence. It has been suggested that a majority of failures in these systems are due to the poor anticipation by the designer of the possible interaction between the system and its environment [43].

  Traditional safety-engineering techniques such as FMEA and fault-tree analysis require that the designer already has full knowledge of potential failure modes of the system. However, with the increasingly tighter integration of such systems, explicitly enumerating all failure modes may be a daunting task. We believe that our approach can be complementary to the existing safety methods; by mapping a design into an environment model and subjecting it to our style of analysis, the designer may discover subtle, unknown interactions between the system and the environment that can lead to a violation of a safety requirement.

- **Side channel attacks**: We have briefly discussed an example of side channel attacks in Section 1.1.2, where the attacker is able to extract information from a program that would be deemed secure under a typical model of input-output behavior. One way to understand these attacks is to treat a program as having multiple representations: one as a traditional machine that produces an output value given a particular input, and another as a physical process that consumes a different amount of time or power depending on the given computation task. When represented in the latter form, the program may be amenable to a different kind of analysis that would reveal its potential security risks.

8.3 Conclusion

The complexity of a modern computer system is far beyond the reach of a single designer's capacity to grasp, and abstraction will continue to be one of the most fundamental techniques in system design. Abstraction, however, is not without its own risks, especially in systems where security is a paramount concern. In an ideal world, one should be able to reason about a security property at a high-level of abstraction, and then expect the same property to be preserved in the final, deployed system. But in practice, the designer is likely to make a series of decisions that inadvertently introduce undesirable behavior into the system, invalidating a property that was established earlier in the development.

This thesis described an approach for addressing this challenge in system design, by suggesting that common security knowledge be encoded into reusable models, and proposing a new composition mechanism for constructing a single, cohesive model of the system across multiple abstraction layers. But having completed this research project, we feel that our proposal is still somewhat unsatisfactory, in that it does not answer the really hard questions: What are the details that can be safely ignored during abstraction, and what are the ones that should be kept? Given a number of alternative ways to construct an abstraction of a system, is one more preferable than the other? If so, is there a systematic method for exploring and evaluating these alternatives?

In general, no abstraction is perfect, and as such, no system can be made invulnerable to all possible attacks. However, there have been previous efforts in trying to answer these types of questions in more restricted settings. Most notably, in their seminal paper, Abadi and Needham suggest a set of design guidelines for constructing a cryptographic protocol that is likely to be resilient against possible attacks (both known and unknown) [3]. The authors of the paper stress that these informal guidelines are meant to be complementary to formal methods—perhaps hinting that there will always be questions that cannot be answered using a formal analysis. Our hope is that by rigorously studying the fundamental issues underlying common security failures, as we have attempted to do in this thesis, our community will be better equipped to develop similar collections of guidelines for secure construction of systems of the future.

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 19–34, 1999.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997.*, pages 36–47, 1997.

[3] Martín Abadi and Roger M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[4] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, pages 3–22, 2000.

[5] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[6] Federal Aviation Administration. Eastern Airlines 375 at Boston: Lessons learned. `http://lessonslearned.faa.gov/ll_main.cfm?TabID=1&LLID=36&LLTypeID=12`. Accessed: 2015-08-02.

[7] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of web security. In *CSF*, pages 290–304, 2010.

[8] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 257–272, 2013.

[9] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *ICSE*, pages 662–671, 2013.

[10] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.)*. Wiley, 2008.

[11] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the small scope hypothesis. `http://sdg.csail.mit.edu/pubs/2002/SSH.pdf`.

[12] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 247–262, 2012.

[13] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

[14] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[15] Shoham Ben-David, Marsha Chechik, and Sebastián Uchitel. Merging partial behaviour models with different vocabularies. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, pages 91–105, 2013.

[16] Yves Bertot and Pierre Casteran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[17] Armin Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, page 88, 2014.

[18] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96, 2001.

[19] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, 2001.

[20] Suresh Chari, Charanjit S Jutla, and Arnab Roy. Universally composable security analysis of oauth v2. 0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[21] Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 609–622, 2015.

[22] Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. Decidability of trace equivalence for protocols with nonces. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 170–184, 2015.

[23] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 51–65, 2008.

[24] Hubert Comon-Lundh and Véronique Cortier. Security properties: Two agents are sufficient. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 99–113, 2003.

[25] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 109–120, 2001.

[26] Nitesh Dhanjani. Abusing the internet of things. https://www.blackhat.com/docs/asia-14/materials/Dhanjani/Asia-14-Dhanjani-Abusing-The-Internet-Of-Things-Blackouts-Freakouts-And-Stakeouts.pdf, 2014. Blackhat Asia.

[27] Edsger W Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[28] Steve M. Easterbrook and Bashar Nuseibeh. Managing inconsistencies in an evolving specification. In *RE*, pages 48–55, 1995.

[29] Daniel Fett, Ralf Küsters, and Guido Schmitz. An expressive model for the web infrastructure: Definition and application to the browser ID SSO system. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 673–688, 2014.

[30] Geri Georg, Indrakshi Ray, Kyriakos Anastasakis, Behzad Bordbar, Manachai Toahchoodee, and Siv Hilde Houmb. An aspect-oriented methodology for designing secure applications. *Information & Software Technology*, 51(5):846–864, 2009.

[31] GrepCode. JDK/jdk/openjdk/6-b14/java.util.Arrays. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Arrays.java`. Accessed: 2015-08-05.

[32] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Browser model for security analysis of browser-based protocols. In *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, pages 489–508, 2005.

[33] Code Hale. A lesson in timing attacks. `http://codahale.com/a-lesson-in-timing-attacks`. Accessed: 2016-1-10.

[34] WG Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[35] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[36] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181, 2014.

[37] Thai Son Hoang, A. K. McIver, Larissa Meinicke, Carroll C. Morgan, A. Sloane, and E. Susatyo. Abstractions of non-interference security: probabilistic versus possibilistic. *Formal Asp. Comput.*, 26(1):169–194, 2014.

[38] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[39] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[40] Internet Engineering Task Force. OAuth Authorization Framework. http://tools.ietf.org/html/rfc6749, 2014.

[41] Daniel Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, 1995.

[42] Daniel Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[43] Daniel Jackson. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[44] Adam Judson. Tamper data plugin for firefox. `https://addons.mozilla.org/en-us/firefox/addon/tamper-data`. Accessed: 2015-03-15.

[45] Dirk O. Keck and Paul J. Kühn. The feature and service interaction problem in telecommunications systems. A survey. *IEEE Trans. Software Eng.*, 24(10):779–796, 1998.

[46] Andrew Kennedy. Securing the .net programming model. *Theor. Comput. Sci.*, 364(3):311–317, 2006.

[47] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[48] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[49] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[50] Barbara Liskov and John V. Guttag. *Program Development in Java - Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

[51] Barbara Liskov and Stephen N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[52] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 18–30, 1997.

[53] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of component and connector models from crosscutting structural views. In *ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 444–454, 2013.

[54] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 16-18, 1994*, pages 79–93, 1994.

[55] Aleksandar Milicevic, Ido Efrati, and Daniel Jackson. αrby - an embedding of alloy in ruby. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 56–71, 2014.

[56] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 609–619, 2015.

[57] Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977.

[58] John C Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of ssl 3.0. In *USENIX Security*, 1998.

[59] Mitre. Common Attack Pattern Enumeration and Classification. http://capec.mitre.org, 2014.

[60] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[61] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve M. Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *ICSE*, pages 54–64, 2007.

[62] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. Expressing the relationships between multiple views in requirements specification. In *ICSE*, pages 187–196, 1993.

[63] Hyun-Kyung Oh and Seung-Hun Jin. The security limitations of SSO in OpenID. In *Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on*, volume 3, pages 1608–1611. IEEE, 2008.

[64] Open Web Application Security Project. OWASP Top Ten Project. http://www.owasp.org/index.php, 2014.

[65] OWASP. Cross-site request forgery (csrf). `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`. Accessed: 2016-1-10.

[66] Aircraft Owners and Pilots Association (AOPA). General aviation safety record: Current and historic. `http://www.aopa.org/About-AOPA/GeneralAviation-Statistics/General-Aviation-Safety-Record-Current-and-Historic`. Accessed: 2015-08-02.

[67] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Formal verification of oauth 2.0 using alloy framework. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 655–659. IEEE, 2011.

[68] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[69] Paypal. Ipn integration guide. `https://developer.paypal.com/webapps/developer/docs/classic/ipn/integration-guide/IPNIntro`. Accessed: 2015-7-15.

[70] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[71] Portswigger. Burp proxy. `http://portswigger.net/burp/proxy.html`. Accessed: 2015-03-16.

[72] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003.

[73] Jan Reineke and Stavros Tripakis. Basic problems in multi-view modeling. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, pages 217–232, 2014.

[74] Joanna Rutkowska. On formally verified microkernels (and on attacking them). `http://theinvisiblethings.blogspot.com/2010/05/on-formally-verified-microkernels-and.html`. Accessed: 2016-1-10.

[75] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[76] Mehrdad Sabetzadeh and Steve M. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *RE*, pages 306–318, 2005.

[77] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[78] Herbert A Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962.

[79] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390. ACM, 2012.

[80] Catchpoint Systems. Anatomy of an http transaction. `http://blog.catchpoint.com/2010/09/17/anatomyhttp`. Accessed: 2016-01-24.

[81] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 160–171, 1998.

[82] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, 2008.

[83] Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In *SIGSOFT FSE*, pages 43–52, 2004.

[84] Rob J. van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Inf.*, 37(4/5):229–327, 2001.

[85] Cybersecurity Ventures. Cybersecurity market report, q4 2015. `http://cybersecurityventures.com/cybersecurity-market-report/`. Accessed: 2016-1-16.

[86] John Viega and Gary McGraw. *Building secure software: How to avoid security problems the right way*. Pearson Education, 2001.

[87] Open Web Application Security Project (OWASP). Category: Attack. `https://www.owasp.org/index.php/Category:Attack`. Accessed: 2015-08-02.

[88] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[89] Xingdong Xu, Leyuan Niu, and Bo Meng. Automatic verification of security properties of oauth 2.0 protocol with cryptoverif in computational model. *Information Technology Journal*, 12(12):2273, 2013.