Multi-representational Security Analysis

Eunsuk Kang*
EECS, UC Berkeley
Berkeley, CA USA
eunsuk.kang@berkeley.edu

Aleksandar Milicevic*
TSE, Microsoft
Redmond, WA USA
almili@microsoft.com

Daniel Jackson
CSAIL, MIT
Cambridge, MA USA
dnj@mit.edu

ABSTRACT

Security attacks often exploit flaws that are not anticipated in an abstract design, but are introduced inadvertently when high-level interactions in the design are mapped to low-level behaviors in the supporting platform. This paper proposes a *multi-representational* approach to security 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. This paper describes Poirot, a prototype implementation of the approach, and reports on our experience on applying Poirot to detect previously unknown security flaws in publicly deployed systems.

CCS Concepts

•Software and its engineering → Formal software verification

Keywords

Security, representation, modeling, verification, composition.

1. INTRODUCTION

Abstraction is a key ingredient of any successful formal analysis. Most systems are too complex to be fully described at once, and so a typical model focuses on a particular aspect of a system at one abstraction layer, and omits details that are deemed irrelevant for the analysis at hand. For example, when reasoning about the business workflow of an online store, the designer may ignore or defer lower-level design decisions, such as the choice of underlying communication protocols, web frameworks, and data structures used.

However, in certain domains, especially those where security is a paramount concern, abstraction can be a double-edged sword. A key observation, noted since the early days of security research [22], is that many security attacks arise due to **discrepancies between the designer's conceptual view of the system and that of an attacker**. While the designer tends to work with one abstraction at a time, the attacker is not bound by such restrictions, and may exploit weaknesses across multiple layers of the system. For example, a malicious user of the online store may chain together an attack that exploits a weakness in a standard browser's handling of cookies, with a lower-level network attack that intercepts packets between two machines.

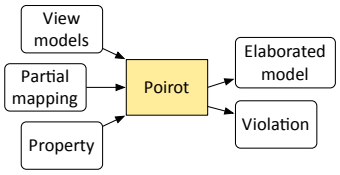

Figure 1: Overview of Poirot.

In this paper, we propose a new methodology called *multi-representational security analysis*. Our approach allows a designer to perform a security analysis to find attacks that involve the behavior of the system across multiple abstraction layers. The analysis can be carried out incrementally: Starting with an abstract model that represents an initial design of the system, the designer can elaborate a part of the model with a choice of representation, transforming the model into a more detailed one.

A key element of our approach is a mechanism for systematically composing independent models of a system in order to enable an end-to-end analysis. However, these models may not be readily amenable to composition due to *abstraction mismatch*: A pair of models may describe a common aspect of the system at different layers of abstraction, using distinct sets of vocabulary terms. To resolve this mismatch, we allow the designer to specify how various parts of the models are related through their *representations*. If some of these relationships are unknown (as often is the case during early design stages), the designer may leave them *unspecified*. Our mechanism then automatically generates candidate mappings that may leave the resulting system vulnerable to attacks, allowing the designer to explore security implications of alternative design decisions.

To demonstrate the feasibility of our approach, we have built a prototype tool called *Poirot*. As shown in Figure 1, the tool takes three inputs from the designer: a desired security property, a set of models that describe different views of a system, and a (potentially partial) mapping that relates parts of those models. Poirot then produces (1) an elabo-

*This work was done when the authors were at MIT.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950356

rated model that captures the end-to-end behavior of the system across those views, and (2) a scenario (if it exists) that demonstrates how the resulting system may violate the given security property.

We have applied Poirot to two publicly deployed systems: (1) IFTTT, a system for composing web services, and (2) HandMe.In, an online system for tracking personal properties. We identified several security flaws in the designs of these systems, some of which could be used to carry out previously unknown attacks.

The contributions of this paper are:

- A *multi-representational* approach to security analysis (outlined in Section 2), in which composition *expands* the range of behaviors to allow for new security violations that lower-level design decisions might introduce;
- A formal specification framework that allows distinct views of a system to be modeled and composed using a *representation mapping* (Section 3);
- An analysis technique for *automatically generating a mapping* that leads to a violation of a property (Section 4);
- An experience report on applying Poirot, our prototype tool (Section 5), to two case studies (Section 6).

2. MOTIVATING EXAMPLE

Consider the task of designing an online store. A typical store offers a variety of services, but we will focus on two basic functions: logging in and adding an item to a shopping cart. A desirable property of the store is the *shopping cart integrity*; that is, every item in a customer's cart must be one of the items that were intentionally added by the customer.
(1) Initial Design and Analysis: Figure 2(a) shows a high-level model of the store system, consisting of three different *processes*: `Store`, which provide `Login` and `Add` services, and `Alice` and `Eve`, customers who interact with the store by invoking these two services. `Login` accepts a user ID and a password, and returns a token that can be used in subsequent interaction with the store. `Add` takes an item ID and a token, and adds the item to the shopping cart of the customer that is identified by the token. In addition, we will assume that Eve is a malicious person, and will deliberately attempt to violate the integrity property by causing the store to add an item of her choice to Alice's cart. Each process is associated with a specification that describes its behavior; for example, the specification on `Store` may state that it only accepts `Add` requests that contain a valid token.

The designer wishes to perform an analysis on this initial model to check whether the system satisfies the integrity property. Let us first consider a conventional analysis in which security violations are found by exploring all possible behaviors of the processes, including the malicious one. Running an analysis tool, such as a model checker, might reveal counterexamples to this property:

- **Ex1**: `Eve` invokes the `Login` service with a victim customer's (i.e., `Alice`) user ID and password, and receives a token that identifies `Alice`. `Eve` then invokes `Add` with this token and an item ID of her choosing, successfully causing the store to add that item to Alice's cart.
- **Ex2**: `Eve` invokes `Login` with her own ID and password, and receives a token that matches Alice's token. `Eve` uses this token to add an item of her choice to Alice's cart.

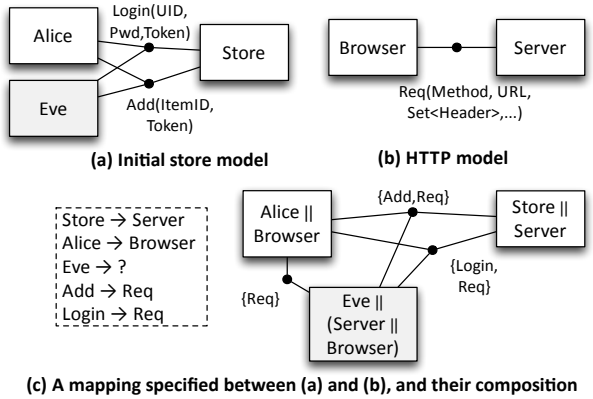

Figure 2: Graphical depictions of system models. A rectangle represents a process; grey ones are considered malicious. A circle represents the participation of a certain type of events by two processes; e.g., in (a), `Alice` and `Store` engage in `Login` events. $p \| q$ corresponds to the composition of p and q. Some events may be associated with multiple representations; e.g., in (c), {`Login`, `Req`} means that every `Login` is realized as an HTTP request.

The designer examines the counterexamples, and derives additional constraints on the behaviors of the individual processes that would prevent those attacks:

- **C1**: `Eve` does not initially have access to the password of another customer (prevents **Ex1**).
- **C2**: The store returns a unique token to each customer in response to `Login` (prevents **Ex2**).

Re-running the analysis with these constraints no longer returns any counterexample, suggesting that the store design ensures the shopping cart integrity. The constraints represent component specifications (on `Store`) and environmental assumptions (`Eve`) that together establish the property.

Note that the level of abstraction in Figure 2(a) is suitable for reasoning about the essential functionality of the system; it omits details that are irrelevant to the business workflow, such as what devices `Store` will be deployed on, how `Login` and `Add` operations are implemented, or the type of data structure used to represent tokens and item IDs. However, as we will see, it may be exactly these details that are exploited by the attacker to undermine the store's integrity.
(2) Design Elaboration: Being satisfied with the initial design of the store, the designer wishes to explore different implementation choices and analyze their impact on the security of the system. In particular, she decides that `Store` is to be deployed on a standard HTTP server, with `Login` and `Add` implemented as HTTP requests.

Figure 2(b) shows an *application-independent* model that describes interactions between a generic HTTP server and a browser, including details about HTTP requests and responses. The designer would like to reuse the domain knowledge captured in this HTTP model by integrating it with the application-specific store model in Figure 2(a). However, these two models, in their current form, are not readily amenable to composition due to *abstraction mismatch*: They describe the system at different levels of abstraction using two distinct sets of vocabulary terms.

The designer's task is then to determine how various parts of the abstract store design are to be realized in terms of

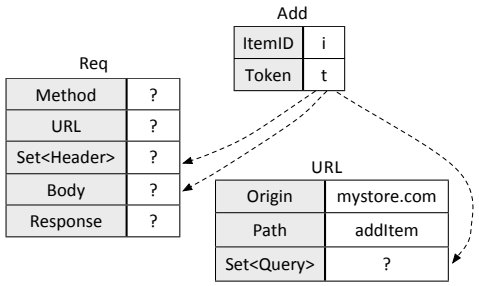

Figure 3: The structures of the `Add` operation, an HTTP request, and a URL. The leftmost column in each structure contains the types of parameters. Dotted edges represent different choices for encoding the token (t) inside a request; "?" represents an unknown value for the parameter.

their counterparts in the HTTP model. For example, as shown in Figure 3, the `Add` operation has a very different structure from an HTTP request, which itself consists of complex data types with their own structures, such as URLs. Thus, realizing `Add` as an HTTP request involves a series of decisions such as:

- Which HTTP method should be used for this operation?
- Which URL should be allocated?
- How should the parameters of `Add` (the item ID and token) be transmitted inside an HTTP request?

Some of these decisions may be fixed as design constraints (e.g., the hostname for the store site), while the rest may be open for the designer to explore. For instance, she may choose to transmit the store token as a browser cookie, or encode it as a query parameter inside the URL. But each of these decisions also carries security implications. Once represented as a cookie, the token may be subject to cookie-related vulnerabilities on the web. On the other hand, when carried inside a URL, the token may be exposed to throughout parts of the browser that operate on the URL. Exploring design alternatives is a challenging task that requires systematic tool support. The space of possible design decisions may be large, prohibiting any kind of manual exploration, and some of those decisions may be irrelevant to the security property being analyzed.

The key idea behind our methodology is to allow the developer to (1) specify a relationship between a pair of abstract and concrete entities by using a *representation mapping*, and (2) express *uncertainty* over design decisions by leaving some part of the mapping as unspecified. Given a partial mapping specification, our analysis automatically explores the space of candidate mappings, and selects ones that lead to a violation of a given security property; such a mapping, if exist, corresponds to a set of insecure design decisions that may leave the resulting system vulnerable to attacks, undermining a property that has previously been established.

(3) Analysis with Partial Mapping: Suppose that the designer provides a mapping indicating that the `Add` operation is to be realized as a GET request with a fixed domain and path (`http://mystore.com/add`), but does not specify how the item or token parameters associated with `Add` should be transmitted as part of the request (similarly, for `Login`). Intuitively, this partial specification yields a space of possible mappings, including ones where the token is encoded inside a header, the request body, or a query parameter in URL, as shown in Figure 3.

Given the store and HTTP models, and the mapping specification, our analysis engine automatically searches for an insecure mapping (if it exists), and uses it to construct an elaborated model that describes the deployment of the store onto an HTTP server, as shown in Figure 2(c). In this final system, the item parameter of `Add` is encoded as a query parameter inside the URL, and the token is encoded as a cookie header in the request. In addition, `Eve` can now act as both a `Browser`, interacting with other servers by sending potentially malicious requests, and as a `Server`, providing its own web pages that may be visited by `Alice` on her browser.

Along with the elaborated model, the analyzer also produces *new* counterexamples to the integrity property—absent from the initial design but introduced by insecure design decisions in the generated mapping:

- **Ex3**: `Alice` visits a fake login page set up by `Eve` (through `Req`) and is tricked into sending her password to `Eve`, who then uses this information to log onto the store and add an item of her choosing to Alice's cart. This counterexample corresponds to a type of phishing attack.

- **Ex4**: `Alice` visits `Eve`'s malicious page, which, when loaded by Alice's browser, triggers an additional request to be sent to a URL of Eve's choosing. In particular, the attacker causes Alice's browser to send an `Add` HTTP request that contains Alice's token as a cookie, and thus, is accepted by the store as valid. This counterexample corresponds to a cross-site request forgery (CSRF).

At this point, the developer may directly modify the model to rule out these attacks; for example, she may add an assumption that `Alice` does not give out her password to an untrusted site, or require that every `Add` request to carry a special CSRF token as an additional proof of authentication. If these modifications are undesirable, she may ask the analyzer to generate an alternative mapping, where the token is mapped to a different part of an HTTP request (which, in turn, may enable different types of attacks). The steps (2) - (3) may be repeated until the analyzer no longer detects a counterexample—implying that the current mapping captures a set of design decisions that preserve the security property in the final, deployed system.

3. SYSTEM MODELING

In this section, we introduce the underlying formalism for constructing models of systems. We describe a simple process algebra in the style of Hoare's CSP [14], where the behavior of a system is defined as a set of *event traces* performed by concurrent processes. We then propose an extension to this framework where certain parts of a system may be assigned *multiple representations*, to enable a new kind of composition where models specified using distinct alphabets are brought together through a *representation mapping*.

3.1 Basic Framework

As the underlying formalism, we adapt the trace-based semantics of CSP [14]. Compared to state-based formalisms, process algebras emphasize communication between different parts of a system, and have been successfully used for security modeling and verification [1, 33, 5].

A system consists of a set of *processes* that interact with each other by together performing various types of *events*. Let P be a set of processes, E a set of events, and T a set of *traces*, each of which is a finite sequence of events. Each process p is associated with an *alphabet* (denoted $\alpha(p) \subseteq E$), which describes the set of events that it may engage in. We say that $t \in T$ is a trace of process p if t describes one possible history of events that p performs up to some moment in time. Then, the overall behavior of p can be defined as the set of all traces that p may perform, and is denoted as $beh(p) \subseteq T$.

A pair of processes, p and q, can be combined into a more complex process that embodies their interaction using the *parallel composition* operator ($p \| q$). The composition rule states that that both p and q must synchronize on events that are common to their alphabets:

$$beh(p \| q) = \{t \in T \mid t \in (\alpha(p) \cup \alpha(q))^* \wedge (t \upharpoonright \alpha(p)) \in beh(p) \wedge (t \upharpoonright \alpha(q)) \in beh(q)\}$$

where $(t \upharpoonright X)$ is the projection of t onto the event set X.

Example. Let $\texttt{Store} \in P$ be a process that behaves like an online store, offering two types of services:

$$\alpha(\texttt{Store}) = \texttt{Login} \cup \texttt{Add}$$
$$\{\langle\rangle, \langle \texttt{login}_{\texttt{Alice}}, \texttt{add}_{\texttt{choc}} \rangle, \langle \texttt{login}_{\texttt{Alice}}, \texttt{add}_{\texttt{toff}}, \texttt{add}_{\texttt{choc}} \rangle\} \subseteq beh(\texttt{Store})$$

Here, $\texttt{Login} \subseteq E$ is the set of events that describe some user logging onto the store; $\texttt{login}_{\texttt{Alice}}$ is a particular event that corresponds to the login action of the user named `Alice`. Three possible traces of `Store` are shown above: an empty trace, one that begins with `Alice` logging in and adding a chocolate (`choc`) to her cart, and another one where a toffee (`toff`) is added before a chocolate.

We are interested in understanding not just how the store behaves on its own, but also how it interacts with customers. Let us introduce a process named `Alice`, who communicates to the store by invoking the latter's services:

$$\alpha(\texttt{Alice}) = \texttt{Login} \cup \texttt{Add}$$
$$\langle \texttt{login}_{\texttt{Alice}}, \texttt{add}_{\texttt{choc}} \rangle \in beh(\texttt{Alice})$$

Consider the above trace that describes `Alice` logging onto the store and adding a chocolate to her cart. Since both the store and Alice are able to synchronize on these events, this trace is also a valid behavior of their composition; i.e.,

$$\langle \texttt{login}_{\texttt{Alice}}, \texttt{add}_{\texttt{choc}} \rangle \in beh(\texttt{Alice} \| \texttt{Store})$$

But suppose that Alice does not like toffees, and so she would never add such an item to her cart on her initiation; thus, event $\texttt{add}_{\texttt{toff}}$ does not appear in any trace of `Alice`. Then, the following is not a trace of the composition:

$$\langle \texttt{login}_{\texttt{Alice}}, \texttt{add}_{\texttt{toff}}, \texttt{add}_{\texttt{choc}} \rangle \notin beh(\texttt{Alice} \| \texttt{Store})$$

even though it is allowed by `Store`, since the two processes cannot synchronize on the second event.

3.2 Multi-Representation Extension

Suppose that our designer wishes to reason about the behavior of the online store when it is deployed as an HTTP server. She is given a process named `Server` (developed independently by a domain expert), which describes the behavior of a *generic* web server that is ready to accept any arbitrary HTTP requests:

$$\alpha(\texttt{Server}) = \texttt{Req}$$

Ideally, she should be able to reuse the knowledge captured in `Server` by integrating it with the `Store` process. The resulting process (named `StoreServer`) would describe a specialized web server that offers the services of an online store.

But `Store` and `Server` engage in two distinct sets of events, and so applying the parallel composition operator would simply result in a system where the two behave completely independent of each other. A new composition mechanism is needed, in order to allow processes from different views of a system to interact with each other. In the next section, we propose a simple extension to the trace-based semantics introduced earlier to achieve this composition.

3.2.1 Events as Sets of Labels

A key idea behind our extension is to allow each event to be associated with multiple representations, by establishing a *label* of an event as a separate notion from the event itself. Let L be a set of *event labels*. Then, **an event is now defined to be a set of labels** ($E = 2^L$), where each label corresponds to one possible description or *representation* of the event. Similarly, the alphabet of a process (p) is redefined to be the set of labels that may appear in any one of p's events ($\alpha(p) \subseteq L$).

Example. Recall the `Store` process from our running example, which can now be reformulated as follows:

$$\alpha(\texttt{Store}) = \texttt{Login} \cup \texttt{Add}$$
$$\langle \{\texttt{login}_{\texttt{Alice}}\}, \{\texttt{add}_{\texttt{choc}}\} \rangle \in beh(\texttt{Store})$$

Note that every event in the above trace contains one label; this corresponds to the basic case where each event has exactly one representation, as in CSP. More complex cases arise when two or more representations, possibly originating from independent models of a system, are associated with the same event in the world. Recall the structure of an HTTP request event:

$$\texttt{req}(m, u, hds, b, resp)$$

where m is the HTTP method, u the URL of the request, hds a set of headers, b the request body, and $resp$ its response. To deploy the store as a web server, the designer may allocate a certain set of URLs for the store operations. For instance, the following URL may be designated for the operation of adding a chocolate:

$$\texttt{url}_{\texttt{choc}} = \texttt{url}(\texttt{origin}_{\texttt{Store}}, \texttt{path}_{\texttt{Add}}, \{\texttt{query}(\texttt{item}, \texttt{choc})\})$$

where $\texttt{origin}_{\texttt{Store}}$ and $\texttt{path}_{\texttt{Add}}$ are particular origin and path values; `query` is a function that constructs a query parameter given some (name, value) pair; and `url` is a function that constructs a URL from an origin, a path, and a set of query parameters. Then, the HTTP request corresponding to this operation can be described as:

$$\texttt{req}_{\texttt{choc}} = \texttt{req}(\texttt{GET}, \texttt{url}_{\texttt{choc}}, \{\texttt{cookie}(\texttt{token}, \texttt{jAx2cE})\}, \_, \_)$$

where the token capturing Alice's identity (value `jAx2cE`) is encoded as a cookie header (the body and response of

the request are irrelevant to our discussion, and denoted by the placeholder variable _). Then, in the process `StoreServer`, an event that results in a chocolate being added to Alice's cart contains two distinct representations:

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

Intuitively, this event can be regarded as an abstract `Add` action or as an HTTP request, depending on the process that engages in the event.

3.2.2 Composition with a Representation Mapping

Let us propose a new composition operator

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

which introduces a relationship between events with distinct labels as specified in the *representation mapping* m and allows p and q to interact through those events. A *representation mapping* is a relation of type $L \times L$, where $(a, b) \in m$ means that "every a event should also be considered as a b." More precisely, this operator requires that whenever p or q performs a, the other process synchronizes by performing b at the same moment; in the composed process, the synchronized event appears as the union of the two original events from p and q.

This new operator is defined similarly to the standard parallel composition ($\|$):

$$\begin{aligned} beh(p \underset{m}{\|} q) = \{t \in T \mid \\ (t \upharpoonright \alpha(p)) \in beh(p) \wedge (t \upharpoonright \alpha(q)) \in beh(q) \wedge \\ \forall\, e \in events(t), a \in e, b \in L \cdot (a, b) \in m \Rightarrow b \in e\} \end{aligned}$$

where the constraint on the second line ensures that every event containing a as a label is assigned b as an additional label in the composite process[1].

Example. To express the relationship between `Store` and `Server`, the following entries may be added in a mapping:

$$m = \{(\texttt{add}_{\texttt{choc}}, \texttt{req}_{\texttt{choc}}), (\texttt{login}_{\texttt{alice}}, \texttt{req}_{\texttt{alice}}), ...\}$$

where $\texttt{req}_{\texttt{choc}}$ and $\texttt{req}_{\texttt{alice}}$ are HTTP encodings of the $\texttt{add}_{\texttt{choc}}$ and $\texttt{login}_{\texttt{alice}}$ operations, respectively. Using this mapping, we can construct a process that behaves like the deployment of the store as a HTTP server:

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

In `StoreServer`, every event containing $\texttt{add}_{\texttt{choc}}$ is accompanied by $\texttt{req}_{\texttt{choc}}$ as an additional label, signifying that the abstract `add` operation is implemented as a concrete HTTP request (and similarly, for $\texttt{login}_{\texttt{alice}}$). For instance, consider the following traces from `Store` and `Server`:

$$\langle\{\texttt{login}_{\texttt{alice}}\}, \{\texttt{add}_{\texttt{choc}}\}\rangle \in beh(\texttt{Store})$$
$$\langle\{\texttt{req}_{\texttt{alice}}\}, \{\texttt{req}_x\}, \{\texttt{req}_{\texttt{choc}}\}\rangle \in beh(\texttt{Server})$$

where $\texttt{req}_x$ is an HTTP request that remains unused for store operations. During the composition step, these two traces are combined into the following trace:

$$\langle\{\texttt{login}_{\texttt{alice}}, \texttt{req}_{\texttt{alice}}\}, \{\texttt{req}_x\}, \{\texttt{add}_{\texttt{choc}}, \texttt{req}_{\texttt{choc}}\}\rangle \in beh(\texttt{StoreServer})$$

Note that the event containing $\texttt{req}_x$ does not require synchronization between the two original processes, since it is not mapped to any other label in m.

3.2.3 Behavioral Implications

The multi-faceted nature of events in our approach enables a *modular*, *open* extension of a process alphabet. Given some process p that interacts with q through events labeled from set $X \subseteq L$, p can be extended to interact with another process, r—possibly through the same events—by assigning additional labels to them from a different set, $Y \subseteq L$; this extension does not require any modification to the existing interaction between p and q. For instance, when `Store` is elaborated into `StoreServer`, customers (e.g., `Alice` and `Eve`) can continue to make use of the store services without being aware of their underlying details as HTTP requests. Similarly, the browser will continue to treat each event in `StoreServer` as an HTTP request served at some designated URL, without necessarily knowing that it implements a particular piece of store functionality.

This type of composition may also introduce new interactions among processes in the world. When an event in process p gains an additional label during composition, it may now be engaged by processes that previously were not able to interact with p. From the security standpoint, some of these interactions may be undesirable, allowing a malicious actor to undermine an existing property of the system.

For instance, let `Browser` be a process that depicts the behavior of a generic browser connected to the web, capable of engaging in arbitrary HTTP requests, including $\texttt{add}_{\texttt{toff}}$:

$$\alpha(\texttt{Browser}) = \texttt{Req}$$
$$\langle\{\texttt{req}_{\texttt{toff}}\}\rangle \in beh(\texttt{Browser})$$

Recall that in Section 3.1, we introduced `Alice` as a process that never engages in an $\texttt{add}_{\texttt{toff}}$ event. `Alice` and `Browser` do not share any labels in their alphabets, and thus cannot influence each other's behavior.

Suppose that during the composition of `Store` and `Server`, $\texttt{add}_{\texttt{toff}}$ is mapped to its HTTP counterpart, $\texttt{req}_{\texttt{toff}}$, inside the representation mapping. Consequently, every event in `StoreServer` that involves adding a toffee to Alice's cart contains two labels; i.e., $e = \{\texttt{add}_{\texttt{toff}}, \texttt{req}_{\texttt{toff}}\}$. Since $\texttt{req}_{\texttt{toff}}$ is a label that appears in $\alpha(\texttt{Browser})$, `StoreServer` and `Browser` may engage in e together; in other words, `Browser` is able to get the store to add a toffee to Alice's cart *indirectly* by exploiting the fact that $\texttt{add}_{\texttt{toff}}$ is implemented as an HTTP request.

4. ANALYSIS

4.1 Declarative Mapping Specification

In our approach, a representation mapping can be specified *declaratively* in the following style:

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

where C is a constraint that describes a relationship between

[1]The projection $(a \upharpoonright X)$ is now defined to operate on sets of labels, by removing labels from a that do not appear in X.

the parameters of a and b; we will call this set comprehension S a *mapping specification*. A candidate mapping m *satisfies* a specification S if C evaluates to true over every tuple in m; i.e., m satisfies S if and only if $m \subseteq S$.

A key feature of this declarative approach is the *partiality* of a specification; that is, the constraint in S may leave unspecified how certain parameters of labels are related to each other.

Example. Consider the following specification of a mapping between labels of type `Add` and `Req`:

$$S_m = \{\texttt{add}(t,i), \texttt{req}(m,u,hds,body,resp) \in L \mid$$
$$m = \texttt{GET} \wedge$$
$$u = \texttt{url}(\texttt{origin}_{\texttt{Store}}, \texttt{path}_{\texttt{Add}}, \{\texttt{query}(\texttt{item}, i)\})\}$$

This specification stipulates that the item ID (i) from the `Add` operation is to be encoded as a query parameter inside the URL, but does not say anything about how the token (t) is encoded, or what should be contained inside the headers, body, or response of the request. Conceptually, this specification describes the relationships between `Add` and `Req` in Figure 3; the parameters that are not explicitly constrained in S correspond to unknown design decisions, and may take on any values from their respective types.

4.2 Property-Guided Search

A partial specification of a mapping yields a space of candidate mappings. Among these candidates, ones that are of particular importance from the analysis perspective are those that may admit "unsafe" traces in the resulting system. Intuitively, these mappings encode *insecure* design decisions, in that they may introduce new behavior into the system that can potentially be exploited for an attack.

Given a mapping specification S, a single analysis can be used to not only find a counterexample trace that violates a given property, but also generate a mapping that permits such a trace to be a valid behavior of the composed system. 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:

$$\exists_{i=1}^{n-1} m_i \subseteq L \times L \cdot (\bigwedge_{j=1}^{n-1} m_i \subseteq S_j) \wedge$$
$$\exists\, Sys \in P \cdot Sys = compose(\{p_1, ..., p_n\}, \{m_1, ..., m_{n-1}\}) \wedge$$
$$\exists\, t \in beh(Sys) \cdot \neg Prop(t)$$

where $compose(X, M)$ returns a process that results from the pairwise composition of the processes in X using the set of mappings in M:

$$compose(\{p_1, ..., p_n\}, \{m_1, ..., m_{n-1}\}) = ((p_1 \underset{m_1}{\parallel} p_2) \underset{m_2}{\parallel} ...) \underset{m_{n-1}}{\parallel} p_n$$

Informally, this problem involves generating a set of mappings that (1) satisfy the user-specified specifications ($S_1, ...S_j$), and (2) when used in the composition of given processes, allow the resulting system (Sys) to produce a trace that leads to the violation of a given property ($Prop$).

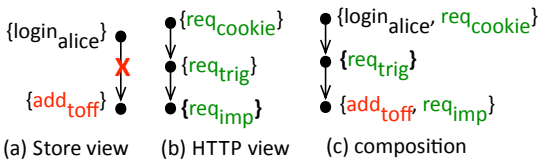

Figure 4: Sample traces from the store example. An event highlighted red is one that results in a violation of a property. An "X" mark on an edge between events e_1 and e_2 means e_2 cannot occur following e_1, rendering the entire trace invalid.

4.3 Example

Let us apply S from Section 4.1 to partially specify a mapping m between `Store` and `Server`. In addition, to model a customer who interacts with the store through a browser, we will use the same mapping to construct a process where Alice's events are represented as HTTP requests:

$$\texttt{StoreServer} = \texttt{Store} \underset{m}{\|} \texttt{Server}$$
$$\texttt{AliceBrowser} = \texttt{Alice} \underset{m}{\|} \texttt{Browser}$$

To model the behavior of an attacker, we will allow `Eve` to act both as a `Browser`, interacting with other servers by sending malicious requests, and as a `Server`, providing its own pages that may be visited by `Alice` on her browser:

$$\texttt{Attacker} = \texttt{Eve} \underset{m'}{\|} (\texttt{Server} \| \texttt{Browser})$$

We want to perform a security analysis against an attacker that behaves non-deterministically in the worst possible manner, and so we will specify a mapping that does not constrain how `Eve` may make use of `Server` and `Browser`:

$$S' = \{a, b \in L \mid \texttt{True}\}$$

Essentially, S' allows the analysis to explore all possible ways of mapping Eve's events onto those of `Browser` or `Server`.

Finally, we can construct the overall system where the attacker interacts with the store and Alice as an attempt to undermine the security of the system:

$$\texttt{StoreSys} = \texttt{StoreServer} \| \texttt{AliceBrowser} \| \texttt{Attacker}$$

Given these processes, the specifications S and S', and the cart integrity property, the analysis engine will attempt to generate m and m' such that when used for composition, the resulting process `StoreSys` allows at least one trace that violates the property.

Analysis Figure 4 shows traces from three different views: (a) the initial store design (`Alice`$\|$`Store`), (b) the generic server-browser architecture (`Server`$\|$`Browser`), and (c) the deployed store system (`StoreSys`). As discussed earlier in Section 3.1, trace (a) is not a valid trace of `Alice`$\|$`Store`, since `Alice` would never perform $\texttt{add}_{\texttt{toff}}$.

Consider trace (b), which describes the following generic HTTP requests in sequence:

- $\texttt{req}_{\texttt{cookie}}(o, c)$: A request that returns a value c and stores it as a cookie at the origin o of the request URL,
- $\texttt{req}_{\texttt{trig}}(u)$: A request that returns an HTML document containing an `img` tag with a URL u in its `src` attribute;
- $\texttt{req}_{\texttt{imp}}(u, cs)$: A request made to a URL u, implicitly generated by the browser after it receives an HTML doc-

ument containing an `img` tag. Also, all cookies (cs) associated with the origin of u are automatically included in the request headers (standard browser behavior).

Our analysis generates a mapping that relates event labels from the store and HTTP models, including ones in Figure 4:

$$m = \{(\texttt{login}_{\texttt{Alice}}, \texttt{req}_{\texttt{cookie}}(\texttt{origin}_{\texttt{Store}}, \texttt{login}_{\texttt{Alice}}.t)), \\ (\texttt{add}_{\texttt{toff}}, \texttt{req}_{\texttt{imp}}(\texttt{url}_{\texttt{toff}}, \{\texttt{add}_{\texttt{toff}}.t\})), ...\}$$

where $l.t$ is the token parameter associated with event label l, and $\texttt{url}_{\texttt{toff}}$ is a URL that has the following structure:

$$\texttt{url}(\texttt{origin}_{\texttt{Store}}, \texttt{path}_{\texttt{Add}}, \{\texttt{query}(\texttt{item}, \texttt{toff})\})$$

The generated mapping indicates that the token given to Alice after the log-in is stored as a cookie in her browser, which then includes the same cookie in subsequent `Add` requests to the store server.

Along with this mapping, the analysis also generates a counterexample trace to demonstrate how the resulting system, `StoreSys`, violates the integrity of the shopping cart (trace (c) in Figure 4). In this sequence, Alice first logs onto the store on her browser, and then visits a malicious site set up by Eve's server through $\texttt{req}_{\texttt{trig}}$. This request is specifically crafted by Eve to trigger Alice's browser to generate a request at the URL ($\texttt{url}_{\texttt{toff}}$) that corresponds to the $\texttt{Add}_{\texttt{toff}}$ event. Since this triggered request ($\texttt{req}_{\texttt{imp}}$) includes Alice's token as a cookie, it is deemed by the store as coming from Alice, causing in the unwanted item to be added to her cart.

5. IMPLEMENTATION

We have implemented a prototype of our approach in a tool called Poirot. The user interacts with the tool by specifying models in Poirot's input language, and running the analysis, which, in turn, leverages a constraint solver for generating mappings and counterexamples. In this section, we briefly describe notable aspects of Poirot; more details about the tool can be found in [20, Chapter 5].

Modeling Language Poirot provides an input language for specifying a system model, security properties, and representation mappings; it is embedded as a domain-specific language in Ruby. Notable features of the language includes a module system for encapsulating domain models, declarative constraints for defining guards and mappings, and additional language constructs for defining component states.

Analysis Engine A Poirot model is translated into Alloy [18], a modeling language based a first-order relational logic. Its backend tool, the Alloy Analyzer, uses an off-the-shelf SAT solver to generate sample instances or check properties of a given model. 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.

Domain Model Library Poirot contains an extensible library of domain models that can be used to elaborate the user's input system model. Domain models encode different types of knowledge, including descriptions of protocols, architecture styles, features, and security threats. For our case studies, we populated the library with a number of domain models that describe different layers of a web system, based on well-known security sources such as OWASP [31] and CAPEC [28]; the models can be categorized as:

- HTTP-related: HTTP protocol, browser scripting, same-origin policy (SOP), cross-origin resource sharing (CORS);
- Network-related: network packets, routing, DNS;
- Authentication mechanisms: symmetric and public-key encryption, OAuth.

The models are approximately 1200 LOC in total, and took around 4 man-months to build; most of the effort was spent on gathering the domain knowledge and ensuring the fidelity and generality of the models. Note that Poirot is not tied to a particular domain, and can be used to model other types of systems, as long as they can be captured in our formalism.

6. CASE STUDIES

We describe our experience on applying Poirot to analyze two publicly deployed systems: IFTTT and Handme.In. Our goal was to answer whether (1) Poirot can be used to find attacks that exploit the details of the system across multiple abstraction layers, and (2) its analysis scales to finding realistic attacks.

6.1 Methodology

Each case study involved the following steps: (1) constructing a model of the system in Poirot, (2) performing an analysis to generate potential attack scenarios and (3) confirming that the scenarios are feasible on the actual system.

In each study, we constructed and analyzed the system model in an incremental fashion, beginning with an abstract design of the system and mapping it to relevant domain models from Poirot's library. Some domain models were employed in both systems (e.g., HTTP protocol, browser scripting), but others were not (e.g, OAuth was relevant only for IFTTT). The complete models for the case studies and the tool are available at http://sdg.csail.mit.edu/projects/poirot.

Each counterexample trace generated by Poirot describes a possible attack on at the *modeling level*. To confirm whether such an attack is feasible in reality, we constructed concrete HTTP requests that correspond to the events, and replayed them from our browser using the TamperData tool [19].

In total, Poirot generated 7 counterexamples for IFTTT, and 8 for HandMe.In; out of these, 4 were confirmed to be feasible attacks (2 for each). The rest were false positives, either due to inaccuracies in our models or generated mappings that do not reflect the actual design of the system. Most of these were caused by our initial misunderstanding of the system behavior; had we ourselves been the developers of the systems, we believe that the false positive rate would have been lower.

6.2 IFTTT

IFTTT is a system that allows a user to connect tasks from different web services using simple conditional statements. The basic concept in IFTTT is a *channel*—a service that exports a number of functions through its API. IFTTT allows a user to make a *recipe*, which consists of a *trigger* and an *action*. Once the recipe is created, each time a trigger is performed, IFTTT automatically executes the corresponding action. For example, one recipe may say that whenever the user is tagged in a Facebook photo, a post containing the photo should be created on the user's Blogger account.

Since IFTTT performs tasks automatically on the user's behalf, possibly accessing her private data, the user must

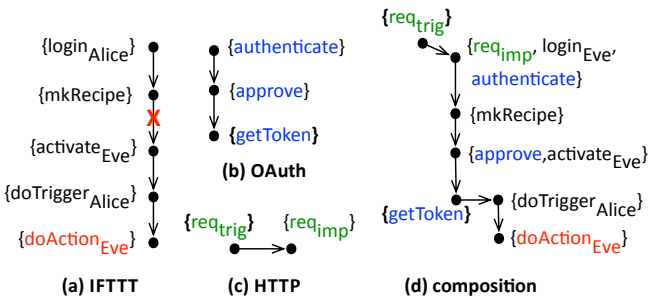

Figure 5: Sample traces from IFTTT models.

give IFTTT a permission to access her accounts on the selected channels; for this purpose, IFTTT employs a third-party authorization protocol called OAuth [16]. For example, before the recipe can take effect, the user must approve IFTTT to (1) access her photos on Facebook, and (2) create new blog posts on her Blogger account. When IFTTT is given a permission to access an account on a channel, we say that the channel has been *activated* for that account.

The goal of this case study was to use Poirot to check whether IFTTT channels could be connected 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 on a channel should only be accessible from the same user's accounts on other channels.

For our study, we constructed a model of the IFTTT workflow (around 140 LOC in Poirot), and composed it against (1) a model of OAuth, to elaborate its authorization process, and (2) the HTTP model from the domain library, to reason about how IFTTT behaves when deployed on a web server. Constructing and analyzing the overall model took us around 1.5 weeks in total. 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 individual web services themselves. Instead, we built archetypal descriptions of channels, triggers, and actions that over-approximate all possible dataflow throughout the system.

We discuss one of the discovered attacks, which exploits details across the three different layers of the system: IFTTT, OAuth, and HTTP.

Information Leakage with Login CSRF One way the attacker (named Eve) may attempt to access private data owned by another user (Alice) is to get IFTTT to link Alice's and Eve's accounts through the same recipe. This way, when a trigger is performed on Alice's account, IFTTT would perform the corresponding action on Eve's account, inadvertently directing data from Alice's account to Eve's.

Figure 5 shows sample traces from the three different models and their composition. In the abstract IFTTT model, which describes its high-level workflow, such an attack is not possible, given an assumption (encoded in the `Alice` process) that when Alice makes a recipe, she activates the associated channels only for her accounts. Thus, a trace where the creation of Alice's recipe (`mkRecipe`) is followed by the activation of a channel for Eve's account ($\texttt{activate}_{\texttt{Eve}}$), as shown in Figure 5(a), is not a valid trace in this model.

Figure 5(b) shows a sequence of events that occurs in a typical OAuth workflow. It begins with a user authenticating herself with an identity provider (e.g., Blogger), which then asks the user to confirm whether she approves access to her account by a third-party (e.g., IFTTT). If so, then the third-party may obtain an access token to her account directly from the identity provider (through `getToken`).

Finally, Figure 5(c) shows a simple trace where one HTTP request ($\texttt{req}_{\texttt{imp}}$) is triggered by another ($\texttt{req}_{\texttt{trig}}$) inside a browser; in Section 4.3, we discussed how this generic sequence may be exploited for a CSRF attack in the context of another system.

In the composition of the three models, an attack on IFTTT becomes possible, as shown in Figure 5(d). It begins with a variant of CSRF (called *login CSRF*), where a victim is unknowingly logged into a site under the attacker's account[2]. This login event has three representations assigned to it by the generated mapping: (1) $\texttt{req}_{\texttt{imp}}$, which is caused indirectly by Eve when Alice visits Eve's site through $\texttt{req}_{\texttt{trig}}$, (2) $\texttt{login}_{\texttt{Eve}}$, which results in Alice getting logged onto a channel site as Eve, and (3) `authenticate`, which fulfills the initial authentication step of an OAuth process.

In the next step, Alice proceeds to make a new recipe with an action on that same channel. Since she is already logged onto the site (albeit as Eve), she is not required to authenticate herself with the action channel during the OAuth workflow; she simply needs to approve IFTTT's access to Blogger. Once IFTTT obtains a token to access Eve's account from the action channel, this account remains associated with the new recipe. As a result, whenever the trigger is performed on Alice's account ($\texttt{doTrigger}_{\texttt{Alice}}$), IFTTT immediately performs the corresponding action on Eve's account ($\texttt{doAction}_{\texttt{Eve}}$), leaking Alice's data to Eve.

As far as we are aware, this was a previously unknown issue with IFTTT. The insecure design decision that enables this attack is the encoding of the login operation on a channel as an HTTP request without protection against CSRF. We confirmed that 4 channel sites on IFTTT were vulnerable to this attack, and notified their developers of the security flaw; 3 of them have since addressed the issue, by adding CSRF protection to their login page.

6.3 HandMe.In

Handme.In (http://handme.in) is a web application designed to facilitate easy recovery of personal items. In a typical workflow, the user purchases a sticker with a unique code written on it, and places the sticker on a physical item to be tracked. If the item is lost, its finder can notify the owner of by entering the code and other relevant information (e.g., an arrangement to return the item in person) on the site. It currently has over 20,000 registered stickers.

The process for the payment of stickers is delegated to Paypal, which offers a service called Instant Payment Notification (IPN). When a customer initiates to purchase a sticker, 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.

For our study, we constructed the models of HandMe.In and Paypal IPN protocols in Poirot (around 200 LOC), and

[2]Normally, login CSRF is considered a minor form of attack, because at most it allows the attacker to access a log of the victim's actions (e.g., search history on Google). In context of IFTTT, however, it can be used to carry out an attack with far more serious consequences, as discussed here.

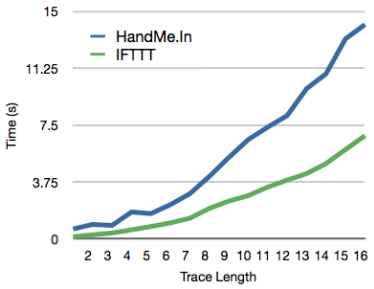

Figure 6: Average analysis times over trace lengths.

composed them together with the HTTP model. Constructing and analyzing the overall model took around 2 weeks. We analyzed the following property of the system: Information about a lost item entered by the finder should be accessible only to the owner of the item. We briefly describe one feasible attack on the system:

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 interacts with the site through a browser. In particular, a web attacker may redirect the user to an IPN site for a sticker that has already been assigned to the attacker; the victim might then unknowingly pay for and receive a sticker that is linked to the attacker, leading to a violation of property (1). This attack combines details from the HandMe.In, Paypal, and HTTP models.

We notified the result of our analysis to the developer, who have since addressed all of the reported security flaws.

6.4 Analysis Performance

We evaluate the scalability of Poirot's analysis over the two case studies. As discussed in Section 5, 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 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. The analyses were performed on a Mac OS X 1.8 GHz with 4G RAM, with the Lingeling [4] SAT solver.

Figure 6 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. Among the counterexamples that were generated, the shortest trace had 4 events, and the longest one 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 a number of examples, we believe that the bounds we used were large enough to capture many common web security attacks.

6.5 Discussion

Threats to Validity There are two potential sources of errors in our case studies: (1) fidelity of the models, and (2) insufficient scope used in analysis (as discussed above). To ensure the accuracy of the HandMe.In system model, we closely worked with its lead developer throughout the entire process; for IFTTT and Paypal IPN, we consulted available documentation on the system, and studied the details of its behavior by inspecting HTTP requests throughout the site. Like in any model-based analysis, however, it is possible that our models inadvertently excluded a detail that would have lead to the discovery of a new counterexample.

Building a faithful model is a challenging task in general, but we believe that it is not an inherent limitation to our composition and analysis approach. For example, to reduce the modeling effort and improve its fidelity, Poirot could be complemented with techniques on extracting models from logs or implementation, some of which have been developed in the context of security [2, 7].

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 (4 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 [16] in order to analyze it, and we were able to reuse the same model for the IFTTT study.

The implementation for each of the two systems was not available, and the public documentation only described the system behavior at the high-level workflow or API level. Thus, we had to deal with some uncertainty about how the system was actually implemented as a web application. Being able to express these unknown information as a partial mapping, and having an automatic analysis to suggest us potentially insecure design decisions, was crucial for our analysis; without this ability, we would need to manually try out possible mappings in an ad-hoc manner.

Incrementality In each of the case studies, we constructed three increments of the system model: (1) composition of an initial design with another abstract model (e.g., IFTTT and OAuth), (2) with an HTTP protocol model, and finally, (3) with a model of browser scripts. Some of the discovered attacks involved only the logic in the abstract design, whereas others also relied on the details in the HTTP and browser models. For example, the attack in Section 6.3 combined a weakness in the business logic of HandMe.In and a browser redirection feature, and thus, was not discovered until the increment (3). But we were also able to find a much simpler attack in (1), which exploited only a flaw in the way HandMe.In assigned codes to its stickers.

Coming up with a complete model of a system at once is often too onerous, and so we believe that being able to discover and address security issues in this incremental manner is crucial to reducing the designer's burden. On the other hand, during each increment, the analysis step became computationally more challenging, as the SAT instance generated by the Alloy Analyzer increased in size. It would be desirable to be able to reuse (where possible) some of the re-

sults from prior analyses; we plan to investigate this problem as a way to improve the scalability of our analysis approach.

7. RELATED WORK

Views Our work was strongly influenced by previous research on *views* in software engineering [10, 17, 30]. In a typical development process, various stakeholders may have differing views on the system, hampering the construction of a single, coherent global model. In the context of security, the attacker can be regarded as one of the stakeholders (with a malicious intent to sabotage the system), exploiting details in a view that differs from that of the designer.

Model merging is an active line of research on techniques and tools for composing independent models. Merging techniques have been developed for various types of models, including architectural views [26], behavioral models [3, 29, 37], database schemas [32], and requirements [34]. Among these, the works on behavioral models are most closely related to our work [3, 29, 37]. A common property guaranteed by their 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 goal is to explore ways in which a property in M_1 may be violated by added behavior from M_2.

Reasoning with Uncertainty Researchers have studied the problem of constructing and analyzing a *partial model*, where certain behavioral or structural aspects of a system are specified to be *unknown*, and an analysis is performed to check properties of the system in the absence of those information [6, 11, 15, 21]. Our work differs from the above approaches in two ways. First, all of these approaches address uncertainty that arises due to delayed decisions about whether a particular feature or characteristic of a system should be included in the final design. In comparison, our approach deals with uncertainty over potential *relationships* between two distinct but possibly overlapping views of a system. Such uncertainty may stem from, for instance, alternative decisions about how a particular abstract operation or data type is to be represented in terms of another, more concrete element. Second, in the existing approaches, uncertainty is expressed over a set of modeling terms with a fixed vocabulary. As a result, they do not handle cases in which new, distinct elements may be introduced and related to parts of an existing model.

In this sense, our work is more closely related to the one by Li, Krishnamurthi, and Fisler [23], where they propose a methodology for composing a system model against independent features and analyzing their potential impact on the system behavior. Like ours, their approach allows new propositions to be introduced into an existing "base" model, giving rise to new behavior and, possibly, a violation of a previously established property. The two approaches differ in the type of composition performed. Their composition involves conjoining a feature with a base model at special, designated states called *interfaces*, and is particularly suited at reasoning about ordered, sequential composition of features. In comparison, our composition involves relating a pair of models across different abstraction layers, and is intended to reason about the impact of design choices on the underlying representation of a system entity.

Security Modeling and Composition A large body of work exists on modeling systems and protocols for security analysis. Most protocol languages describe a system in terms of abstract agents and messages between them, and are not designed for elaborating their underlying representations [1, 24, 36]. Several *compositional security* frameworks have been proposed as a way of establishing the end-to-end security of a system by combining the properties of individual component models [9, 12, 25]. In these approaches, composition involves bringing together two parallel processes that communicate through a fixed interface at the same level of abstraction (e.g., a server and a client). In comparison, our approach involves composing processes that partly or entirely overlap with each other, in that they represent the same entity at different levels of abstraction (e.g., abstract store and concrete server on which it is deployed).

Another related work is Georg and her colleagues' work on an aspect-oriented approach to security modeling and analysis [13]. 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 does not allow, for example, a more complex mapping that relates the *structures* of two elements (e.g., encoding Add as Req).

8. LIMITATIONS AND CONCLUSIONS

In our approach, all events are treated as *atomic* entities. 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 would need to be extended to allow an event to be mapped to a *sequence* of events.

Poirot currently allows the user to specify and check *trace properties*—a type of properties 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 [8]. 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 would need to be extended to perform a higher-order reasoning where *sets* of traces are explored at once.

Given a partially specified m, our analysis generates mappings that lead to security violations. It may also be possible to instead generate a mapping that guarantees that the resulting system *satisfies* a given property; this would involve reasoning about each candidate mapping over all possible traces of a system, and thus require a higher-order analysis. As a next step, we plan to incorporate techniques from software synthesis [27, 35] to provide this capability in Poirot.

Acknowledgement We thank Stéphane Lafortune, Matt McCutchen, Joseph Near, Stavros Tripakis, and our reviewers for their insightful comments and suggestions. This work was supported in part by the NSF Award CRD-0707612, and by the Singapore University of Technology and Design.

9. REFERENCES

[1] M. Abadi and A. 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.

[2] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: automatic extraction of web authentication protocols from implementations. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[3] S. Ben-David, M. Chechik, and S. 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.

[4] A. 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.

[5] B. 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.

[6] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *11th International Conference on Computer Aided Verification, CAV 1999, Italy*, pages 274–287, 1999.

[7] K. Z. Chen, W. He, D. Akhawe, V. D'Silva, P. Mittal, and D. Song. ASPIRE: iterative specification synthesis for security. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[8] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[9] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[10] S. M. Easterbrook and B. Nuseibeh. Managing inconsistencies in an evolving specification. In *RE*, pages 48–55, 1995.

[11] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland*, pages 573–583, 2012.

[12] D. Garg, J. Franklin, D. K. Kaynar, and A. Datta. Compositional system security with interface-confined adversaries. *Electr. Notes Theor. Comput. Sci.*, 265:49–71, 2010.

[13] G. Georg, I. Ray, K. Anastasakis, B. Bordbar, M. Toahchoodee, and S. H. Houmb. An aspect-oriented methodology for designing secure applications. *Information & Software Technology*, 51(5):846–864, 2009.

[14] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[15] M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *10th European Symposium on Programming Languages and Systems, ESOP 2001, Genova, Italy*, pages 155–169, 2001.

[16] Internet Engineering Task Force. OAuth Authorization Framework. http://tools.ietf.org/html/rfc6749, 2014.

[17] D. Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, 1995.

[18] D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[19] A. Judson. Tamper data plugin for firefox. https://addons.mozilla.org/en-us/firefox/addon/tamper-data. Accessed: 2015-03-15.

[20] E. Kang. *Multi-Representational Security Modeling and Analysis*. PhD thesis, MIT, 2016.

[21] O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.

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

[23] H. C. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *10th Symposium on Foundations of Software Engineering, South Carolina, USA*, pages 89–98, 2002.

[24] G. 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.

[25] H. Mantel. On the composition of secure systems. In *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*, pages 88–101, 2002.

[26] S. Maoz, J. O. Ringert, and B. 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.

[27] A. Milicevic, J. P. Near, E. Kang, and D. 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.

[28] Mitre. Common Attack Pattern Enumeration and Classification. http://capec.mitre.org, 2014.

[29] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE*, pages 54–64, 2007.

[30] B. Nuseibeh, J. Kramer, and A. Finkelstein. Expressing the relationships between multiple views in requirements specification. In *ICSE*, pages 187–196, 1993.

[31] Open Web Application Security Project. OWASP Top Ten Project. http://www.owasp.org/index.php, 2014.

[32] R. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003.

[33] P. Y. A. Ryan and S. A. Schneider. *Modelling and*

analysis of security protocols. Addison-Wesley-Longman, 2001.

[34] M. Sabetzadeh and S. M. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *RE*, pages 306–318, 2005.

[35] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415, 2006.

[36] F. J. Thayer, J. C. Herzog, and J. 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.

[37] S. Uchitel and M. Chechik. Merging partial behavioural models. In *SIGSOFT FSE*, pages 43–52, 2004.