Synthesis of Property-Preserving Mappings

Eunsuk Kang *1, Stéphane Lafortune †2, and Stavros Tripakis ‡3

1University of California, Berkeley & University of Michigan
2University of Michigan
3University of California, Berkeley & Aalto University

May 16, 2017

Abstract

System development often involves decisions about how a high-level design is to be implemented using primitives from a low-level platform. Certain decisions, however, may introduce undesirable behavior into the resulting implementation, possibly leading to a violation of a desired property that has already been established at the design level. In this paper, we introduce the problem of synthesizing a property-preserving mapping: A set of implementation decisions ensuring that a desired property is preserved from a high-level design into a low-level platform implementation. We provide a formalization of the synthesis problem and propose a technique for synthesizing a mapping based on symbolic constraint search. We describe our prototype implementation, and a case study demonstrating the application of our technique to synthesizing secure mappings for OAuth, a popular family of authorization protocols on the web.

1 Introduction

When building a complex software system, one begins by coming up an abstract design, and then constructs an implementation that conforms to this design. In practice, there are rarely enough time and resources available to build an implementation from scratch, and so this process often involves reuse of an existing platform—a collection of generic components, data structures, and libraries that are used to build an application in a particular domain.

The benefits of reuse also come with potential risks. A typical platform exhibits its own complex behavior, including subtle interactions with the environment that may be difficult to anticipate and reason about. Typically, the developer must work with the platform as it exists, and is rarely given the luxury of being able to modify it and remove unwanted features. For example, when building a web application, a developer must work with a standard browser and take into account all of its features and security vulnerabilities. As a result, achieving an implementation that perfectly conforms the design may be too difficult in practice. Worse, the resulting implementation may not necessarily preserve desirable properties that have already been established in the design.

These risks are especially evident in applications where security is a major concern. For example, OAuth 2.0, a popular authorization protocol subjected to rigorous and formal analysis at an abstract level [12, 35, 46], has been shown to be vulnerable to attacks when implemented on a web browser or a mobile device [42, 45, 13]. Many of these vulnerabilities are not due to simple programming errors: They arise from logical flaws that involve a subtle interaction between the protocol logic and the details of the underlying platform. Unfortunately, OAuth itself does not explicitly guard against these flaws, since it is intended to be a generic,

*eunsuk.kang@berkeley.edu
†stephane@umich.edu
‡stavros@eecs.berkeley.edu
abstract protocol that deliberately omits details about potential platforms. On the other hand, anticipating and mitigating against these risks require an in-depth understanding of the platform and security expertise, which many developers do not possess.

This paper proposes an approach to help developers overcome these risks and achieve an implementation that preserves desired properties. In particular, we formulate this task as the problem of automatically synthesizing a property-preserving mapping: A set of implementation decisions ensuring that a desired property is preserved from a high-level design into a low-level platform implementation.

Our approach builds on the prior work of Kang et al. [31], which proposes a modeling and verification framework for reasoning about security attacks across multiple levels of abstraction. The central notion in this framework is that of a mapping, which captures a developer’s decisions about how abstract system entities are to be realized in terms of their concrete counterparts. We extend their formal framework with the problem of synthesizing a property-preserving mapping, and propose a novel, algorithmic technique for performing this synthesis task.

We have built a prototype implementation of the synthesis technique. Our tool accepts a high-level design model, a desired system property (both specified by the developer), and a model of a low-level platform (built and maintained separately by a domain expert). The tool then produces a mapping (if any) that ensures that the resulting platform implementation preserves the given property.

As a case study, we have successfully applied our tool to synthesize property-preserving mappings for two different variants of OAuth onto a model of an HTTP platform. Our preliminary results are promising: The implementation decisions captured by our synthesized mappings describe effective mitigations against some of the common vulnerabilities that have been found in deployed OAuth implementations [42, 45].

The contributions of this paper include:

- A formulation of the mapping synthesis problem, a novel approach for ensuring the preservation of a property between a high-level design and its implementation (Section 4);
- A technique for automatically synthesizing mappings based on symbolic constraint search (Section 5); and
- A prototype implementation of the synthesis technique, and a case study demonstrating the feasibility of this approach (Section 6).

The rest of the paper is structured as follows. We begin by introducing a small, illustrative example (Section 2), and then provide a brief overview of the mapping-based framework proposed by Kang et al. [31] (Section 3). We describe our extension of this framework with the synthesis problem (Section 4), a technique for automatically synthesizing a mapping (Section 5), and our case study on OAuth (Section 6). We conclude with a discussion of related work (Section 7) and the limitations of our current approach (Section 8).
2 Illustrative Example

Let us introduce a running example that we will use throughout the paper. Consider the simple state machine in Figure 1(a), consisting of three interacting processes, Alice, Bob, and Eve. In this system, Alice wishes to communicate messages to Bob and Eve, but is willing to share its secrets only with Bob. Alice has access to two separate communication channels (represented by event labels writeBob and writeEve). Alice behaves as follows: It first non-deterministically selects a message to be sent \((m)\) from some set \(\text{Msg}\). If the selected message is not a secret (represented by constant \(\text{secret} \in \text{Msg}\)), Alice sends \(m\) to either Bob or Eve over the corresponding channel (i.e., by performing writeBob or writeEve). If, however, the chosen message is a secret, Alice sends \(m\) only to Bob.

Suppose that Eve is a malicious character whose goal is to learn the secret shared between Alice and Bob. One desirable property of the system is that Eve should never be able to learn the secret; this property can be stated as the LTL formula \(\neg \Diamond (\text{SecretLearned})\). It can be observed that the composition of the three processes, \(\text{Alice} || \text{Bob} || \text{Eve}\), satisfies this property, since Alice, by design, never sends a secret over the channel to Eve.

Now consider the model in Figure 1(b), which describes communication between a pair of processes over an encrypted public channel (represented by event label \(\text{encWrite}(m, k)\)), where \(m\) is the message being sent, and \(k\) is the key used to encrypt the message. Every process is associated with a fixed value called knows, which represents the set of keys that it has access to. In its behavior, Sender non-deterministically chooses a message and a key to encrypt it with. Receiver, upon receiving the message, is only able to read it only if it knows the key that \(m\) is encrypted with \((k)\), or if \(m\) has not been encrypted at all (i.e., \(k = \text{none}\)). In this particular model, we represent Receiver being unable to read message \(m\) as the absence of a corresponding transition.

Suppose that we wish to reason about the behavior of the abstract communication system from Figure 1(a) when it is implemented over the public channel in 1(b). In particular, in the low-level implementation, Bob and Eve are required to share the same channel (\(\text{encWrite}\)), no longer benefitting from the separation provided by the abstraction in Figure 1(a). The developer’s task is to decide how the abstract events writeBob and writeEve are to be implemented as \(\text{encWrite}\).

Does the property of the abstract communication hold in every possible implementation? If not, which decisions will ensure that secret messages remain protected from Eve in the resulting implementation? In this paper, we describe how these questions can be formulated and tackled as the problem of synthesizing a property-preserving mapping between a pair of models that depict a high-level design and a low-level platform.

3 Mappings in System Design

Our synthesis approach builds on the modeling and verification framework proposed in [31], which is designed to allow reasoning about behavior of processes across multiple abstraction layers. In this framework, a trace-based semantic model (based on CSP [23] is extended to represent events as sets of labels, and includes a new composition operator based on the notion of mappings, which relate event labels from one abstraction layer to another. In this section, we present the essential elements of this framework.

3.1 Modeling Framework

Events, traces, and processes Let \(L\) be a potentially infinite set of labels. An event is a finite set of labels (i.e., \(e \in 2^L\)), and a trace is a finite or infinite sequence of events \((t \in T(L) = (2^L)^* \cup (2^L)^\ast)\). The empty trace is denoted by \(\langle \rangle\), and the trace consisting of the single event \(e\) is denoted \(\langle e \rangle\). If \(t\) and \(t'\) are traces, then \(t \cdot t'\) is the trace obtained by concatenating \(t\) and \(t'\). Note that \(\langle \rangle \cdot t = t \cdot \langle \rangle = t\) for any trace \(t\).

The events of a trace \(t\), written \(\text{events}(t)\), is the set of all events appearing in \(t\).

Let \(t\) be a trace over set of labels \(L\), and let \(A \subseteq L\) be a subset of \(L\). The projection of \(t\) onto \(A\), denoted \(t_A\), is the trace obtained by replacing each \(e \in \text{events}(t)\) that is not in \(A\) by \(\langle \rangle\), i.e.,

\[
\text{projection of } t \text{ onto } A = \{ \langle \rangle \cdot t' : t' \in T(L) \land \text{events}(t') \subseteq A \}.
\]
t \upharpoonright A$, is defined as follows\(^1\)

\[
\emptyset \upharpoonright A = \emptyset \quad ((e) \cdot t) \upharpoonright A = \begin{cases} (e \cap A) \cdot (t \upharpoonright A) & \text{if } e \cap A \neq \emptyset \\ (t \upharpoonright A) & \text{otherwise} \end{cases}
\]

A process is defined as a set of traces. The alphabet of process $P$, denoted $\alpha(P)$, is the set of all labels appearing in $P$.

A pair of processes $P$ and $Q$ synchronize with each other by performing $e_1$ and $e_2$, respectively, if these two events share at least one label. In their parallel composition, denoted $P\parallel Q$, this synchronization is represented by a new event $e'$ that is constructed as the union of $e_1$ and $e_2$ (i.e., $e' = e_1 \cup e_2$). Formally, the parallel composition is defined as:

\[
P\parallel Q = \{ t \in T(\alpha(P) \cup \alpha(Q)) \mid (t \upharpoonright \alpha(P)) \in P \land (t \upharpoonright \alpha(Q)) \in Q \land \forall e \in \text{events}(t) : \text{cond}(e) \}
\]

(1)

where $\text{cond}(e)$ is defined as:

\[
\text{cond}(e) \equiv e \subseteq \alpha(P) \lor e \subseteq \alpha(Q) \lor (\exists a \in e : a \in \alpha(P) \land \alpha(Q))
\]

The first line of (1) is similar to the original definition of parallel composition in CSP \cite{17}, stating that if we take a trace $t$ in the composite process and ignore labels that appear only in $Q$, then the resulting trace must be a valid trace of $P$ (and symmetrically for $Q$). The condition on the second line is added to ensure that every event performed together by $P$ and $Q$ contains at least one common label shared by both processes.

Mapping-based composition A mapping is a partial function $m : L \to L$ used to introduce a relationship between a pair of distinct labels. Informally, $m(a) = b$ stipulates that every event that contains $a$ as a label is to be assigned $b$ as an additional label. We use the notation $a \mapsto_m b$ as a shorthand to denote $m(a) = b$.

The mapping composition allows a pair of processes to interact with each other over distinct labels. Given processes $P$ and $Q$ and and mapping $m : L \to L$, the composition $(P\parallel_m Q)$ is defined as\(^2\)

\[
(P\parallel_m Q) = \{ t \in T(\alpha(P) \cup \alpha(Q)) \mid (t \upharpoonright \alpha(P)) \in P \land (t \upharpoonright \alpha(Q)) \in Q \land \forall e \in \text{events}(t) : \text{cond}'(e) \land (\forall a \in e, b \in L : m(a) = b \Rightarrow b \in e) \}
\]

(2)

where $\text{cond}'(e)$ is defined as:

\[
\text{cond}'(e) \equiv \text{cond}(e) \lor (\exists a \in e \cap \alpha(P), b \in e \cap \alpha(Q) : m(a) = b \lor m(b) = a)
\]

The first two lines of the definition (2) is similar to (1) above. The additional disjunct in $\text{cond}'(e)$ allows $P$ and $Q$ to synchronize (even when they do not share any label) if at least one pair of their labels are mapped to each other in $m$. The last line of (2) ensures that every label $a$ that has been mapped to another label $b$ in $m$ does not appear alone in an event without $b$. Note that when $m$ is empty, the composition operator produces the same process as the parallel composition; i.e., $(P\parallel_m Q) = P\parallel Q$ for $m = \emptyset$.

$P$ is typically a high-level, or abstract, model of an application design, while $Q$ is a model of a low-level platform on which $P$ is to be implemented ($P$ and $Q$ may themselves consist of several processes). In most cases, the two processes describe system artifacts that are built independently from each other, and thus do not have any common labels. The mapping $m$ captures decisions on how events from $P$ are to be realized in terms of their counterparts in $Q$: $(P\parallel_m Q)$ describes the result of implementing $P$ on $Q$ using these decisions.

Example. Let $P$ and $Q$ be the abstract and public channel communication models from Figure 1 respectively. For simplicity, we restrict the domains of parameters (i.e., messages and keys) inside event labels to be finite, as follows:

\[
\text{Msg} = \{\text{public, secret}\} \quad \text{Key} = \{\text{none, key}_X, \text{key}_Y\}
\]

\(^1\)Note that this operator has a different meaning from the standard projection in CSP.

\(^2\)Note that this definition is different from the one in \cite{31}, as it refines the original definition to exclude the undesirable case in which events $e_1$ and $e_2$ from $P$ and $Q$ may be synchronized into $e'$ as their union even when they do not share any label.
This, in turn, results in a finite set of possible labels (and thus, a finite number of possible mappings):

\[ L_P = \{ \text{writeBob}(\text{public}), \text{writeBob}(\text{secret}), \]
\[ \text{writeEve}(\text{public}), \text{writeEve}(\text{secret}) \} \]
\[ L_Q = \{ \text{encWrite}(\text{secret}, \text{none}), \text{encWrite}(\text{secret}, \text{key}_X), \]
\[ \text{encWrite}(\text{secret}, \text{key}_Y), \text{encWrite}(\text{public}, \text{none}), \]
\[ \text{encWrite}(\text{public}, \text{key}_X), \text{encWrite}(\text{public}, \text{key}_Y) \} \]

Suppose that we decide on a simple implementation scheme where the abstract messages (\text{writeBob} and \text{writeEve}) are transmitted over the public channel unencrypted; this decision can be represented as a mapping where each abstract label is mapped to \text{encWrite} with \( k = \text{none} \):

\[ \text{writeBob}(\text{secret}), \text{writeEve}(\text{secret}) \mapsto_{m_1} \text{encWrite}(\text{secret}, \text{none}) \]
\[ \text{writeBob}(\text{public}), \text{writeEve}(\text{public}) \mapsto_{m_1} \text{encWrite}(\text{public}, \text{none}) \]

Then, traces of process \((P \parallel_{m_1} Q)\) may include an event that consists of the following two labels:

\[ e = \{ \text{writeBob}(\text{secret}), \text{encWrite}(\text{secret}, \text{none}) \} \]

Informally, the labels of an event correspond to its representations at multiple abstraction layers; in this case, \( e \) is considered both a \text{writeBob} and an \text{encWrite} event.

A singleton event \( e' = \{ \text{writeBob}(\text{secret}) \} \) cannot appear in any trace of \((P \parallel_{m_1} Q)\), since the mapping \( m_1 \) requires that each event containing \text{writeBob}(\text{secret})\) include \text{encWrite}(\text{secret}, \text{none})\) as an additional label.

### 3.2 Mapping Verification

A high-level process \( P \) is typically associated with a specification \( S \) that describes a desired property of the system. In this paper, we assume \( S \) to be expressible as a trace property \( X \) (i.e., \( S \) is a set of traces containing desirable system behavior). A typical verification problem involves, given a process \( P \) and a property \( S \), checking whether \( P \) satisfies \( S \) (i.e., \( P \models S \)). In this framework, an additional verification problem is defined as follows:

**Problem 1 (Mapping Verification)** Given processes \( P \) and \( Q \), mapping \( m \), and property \( S \), check whether \((P \parallel_{m} Q) \models S\).

We say a mapping \( m \) is valid (w.r.t. \( P, Q \) and \( S \)) if and only if \((P \parallel_{m} Q) \models S\). When solving the mapping verification problem, we may rely on the assumption that \( P \) alone satisfies the specification, i.e., that \( P \models S \) holds (which could be checked beforehand). Note that this problem is conceptually an \( \exists \)-problem, as it can be stated as finding a witness trace \( t \) to formula \( \exists t : t \in (P \parallel_{m} Q) \land t \notin S \).

**Example.** Recall the mapping \( m_1 \) from Section 3.1, where abstract messages are transmitted as unencrypted public messages. Let \( \text{Alice}' \) and \( \text{Eve}' \) be processes that represent the implementation of \( \text{Alice} \) as \( \text{Sender} \) using \( m_1 \) (i.e., \( \text{Alice}\parallel_{m_1} \text{Sender} \)), and \( \text{Eve} \) as \( \text{Receiver} \), respectively. The following \( t_1 \) and \( t_2 \) are valid traces of these two processes:

\[ t_1 = \langle \{ \text{writeBob}(\text{secret}), \text{encWrite}(\text{secret}, \text{none}) \} \rangle \in \text{Alice}' \]
\[ t_2 = \langle \{ \text{writeEve}(\text{secret}), \text{encWrite}(\text{secret}, \text{none}) \} \rangle \in \text{Eve}' \]

When composed in parallel, \( \text{Alice}' \) and \( \text{Eve}' \) may interact through the above two events, since they share the common label \text{encWrite} (secret, none). According to the definition (1) from Section 3.1, the following is a valid trace of their parallel composition:

\[ \langle \{ \text{writeBob}(\text{secret}), \text{writeEve}(\text{secret}), \text{encWrite}(\text{secret}, \text{none}) \} \rangle \]
\[ \in (\text{Alice}'\parallel\text{Eve}') \]

Note that in Figure 1(a), this event results in the transition of \( \text{Eve} \) into state \text{SecretLearned}, leading to a

\[ ^3 \text{However, the verification methodology proposed in [31] is generic, and can be used for any types of properties, assuming the availability of a verification procedure that can check whether } P \models S. \]
violation of the property $\neg\Diamond(\text{SecretLearned})$. This can be seen as an example of abstraction violation: As a result of design decisions in $m_1$, $\text{sendBob}$ and $\text{sendEve}$ now share the same underlying representation ($\text{enc}$), and $\text{Eve}'$ is now able to engage in an event that was not previously available to it in the original abstract model.

4 Mapping Synthesis

Building on the formal model in Section 3, we propose the novel problem of synthesizing a mapping that preserves a desired property $S$ between $P$ and $Q$.

**Problem 2 (Mapping Synthesis)** Given processes $P$ and $Q$, and property $S$, find, if it exists, a mapping $m$ such that $(P \parallel_m Q) \models S$.

Note that this problem can be stated as a $\exists\forall$ problem; that is, finding a witness $m$ to the formula $\exists m : \forall t : t \in (P \parallel_m Q) \Rightarrow t \in S$.

Instead of synthesizing $m$ from scratch, the developer may often wish to express her partial system knowledge as a given constraint, and ask the synthesis tool to generate a mapping that adheres to this constraint. For instance, given labels $a, b, c \in L$, one may express a constraint that $a$ must be mapped to either $b$ or $c$ as part of every valid mapping; this gives rise to two possible candidate mappings, $m_1$ and $m_2$, where $m_1(a) = b$ and $m_2(a) = c$.

Formally, let $M$ be the set of all possible mappings between labels $L$. A *mapping constraint* $C \subseteq M$ is a set of mappings that are considered legal candidates for a final, synthesized valid mapping. Then, the problem of synthesizing a mapping given a constraint can be formulated as follows:

**Problem 3 (Generalized Mapping Synthesis)** Given processes $P$ and $Q$, property $S$, and mapping constraint $C$, find, if it exists, a mapping $m$ such that $m \in C$ and $(P \parallel_m Q) \models S$.

Note that Problem 2 is a special case of Problem 3 where $C = M$.

The synthesis problem can be further generalized to one that involves synthesizing a constraint that contains a set of valid mappings:

**Problem 4 (Mapping Constraint Synthesis)** Given a pair of processes $P$ and $Q$, property $S$, and mapping constraint $C$, generate a non-empty constraint $C'$ such that $C' \subseteq C$, and for every $m \in C'$, $(P \parallel_m Q) \models S$.

If it exists, $C'$ yields a set of valid mappings that adhere to the given constraint $C$ (i.e., $C'$ is a stronger constraint than $C$). We call such constraint $C'$ *valid* with respect to $P, Q, S$ and $C$. A procedure for solving Problem 4 can in turn be used to solve Problem 2. Having generated constraint $C'$, we pick some mapping $m$ such that $m \in C'$.

In practice, it is desirable for a synthesized constraint to be as large as possible while still being valid, as it provides more implementation choices (i.e., possible mappings). The problem of synthesizing an optimal mapping constraint is defined as:

**Problem 5 (Optimal Constraint Synthesis)** Given a pair of processes $P$ and $Q$, property $S$, and given constraint $C$, generate a constraint $C'$ such (1) $C'$ is valid with respect to $P, Q, S,$ and $C$, and (2) there exists no other constraint $C''$ such that $C''$ is valid and $C' \subseteq C''$.

Note that $C'$, if found, is a local optimum. In general, there may be multiple local optimal constraints for given $P, Q, S,$ and $C$.

**Example.** Recall our running example from Figure 1. Suppose that there are two Receiver processes with the identical behavior ($\text{Receiver}_X$ and $\text{Receiver}_Y$), except they are assigned unique decryption keys:

$$\text{knows(Receiver}_X) = \{\text{key}_X\} \quad \text{knows(Receiver}_Y) = \{\text{key}_Y\}$$

Furthermore, let $\text{Bob}'$ and $\text{Eve}'$ be processes that result from implementing Bob and Eve as $\text{Receiver}_X$ and
Receiver, respectively. Then, the following $m_2$ is a valid mapping that preserves the property $\neg \Diamond (\text{SecretLearned})$:

- $\text{writeBob} (\text{secret}) \mapsto \text{encWrite} (\text{secret}, \text{key}_X)$
- $\text{writeBob} (\text{public}) \mapsto \text{encWrite} (\text{public}, \text{key}_X)$
- $\text{writeEve} (\text{secret}) \mapsto \text{encWrite} (\text{secret}, \text{key}_X)$
- $\text{writeEve} (\text{public}) \mapsto \text{encWrite} (\text{public}, \text{key}_Y)$

That is, if the secret message is always encrypted with the key assigned to Bob, Eve will never be able to read it. The following $m_3$ is also a valid mapping:

- $\text{writeBob} (\text{secret}) \mapsto \text{encWrite} (\text{secret}, \text{key}_X)$
- $\text{writeBob} (\text{public}) \mapsto \text{encWrite} (\text{public}, \text{none})$
- $\text{writeEve} (\text{secret}) \mapsto \text{encWrite} (\text{secret}, \text{key}_X)$
- $\text{writeEve} (\text{public}) \mapsto \text{encWrite} (\text{public}, \text{none})$

since Eve being able to read public messages does not violate the property. Thus, the developer may choose either $m_2$ or $m_3$ to implement the abstract channel and ensure that secret remains protected from Eve. In other words, $C_1 = \{ m_2, m_3 \}$ is a valid (but not necessarily optimal) mapping constraint with respect to the desired property.

Furthermore, $C_1$ is arguably more desirable than another constraint $C_2 = \{ m_2 \}$, since the former gives the developer more implementation choices than the latter does.

### 5 Synthesis Technique

This section presents a method for representing mappings symbolically, and a technique for synthesizing optimal mapping constraints.

#### 5.1 Symbolic Mapping Representation

Assuming that the set of event labels $L$ is finite, one possible way to represent a mapping is by explicitly listing all of the entries in the function. An alternative representation is one where mappings are represented symbolically as logical expressions over variables that correspond to parameters inside labels. The symbolic representation has the following advantages: (1) it provides a succinct representation of implementation decisions to the developer (which is especially important as the size of the mapping grows large) and (2) it allows the user to specify partial implementation decisions (i.e., given constraint $C$) in a declarative manner.

In particular, we use a grammar-based approach, where the space of candidate mapping constraints is restricted to expressions that can be constructed using a syntactic grammar $[4]$. For the systems that we have studied, we use the following grammar $[4]$:

- $\text{Term} := \text{Var} \mid \text{Const}$
- $\text{Assign} := (\text{Term} = \text{Term})$
- $\text{Expr} := \text{Assign} \mid \text{Assign} \Rightarrow \text{Assign} \mid \text{Expr} \land \text{Expr}$

where $\text{Var}$ is a set of variables that represent parameters inside a label, and $\text{Const}$ is the set of constant values. Intuitively, this grammar captures implementation decisions that involve assignments of parameters in an abstract label to their counterparts in a concrete label (represented by the equality operator “=”). A logical implication is used to construct a conditional assignment of a parameter.

Each expression $E \in \text{Expr}$ is a formula that describes a relationship between a pair of labels (e.g., $(a, b)$) that may appear in a mapping (i.e., $m(a) = b$). To define what it means for a label pair to satisfy an expression, let us introduce the notion of signatures, each of which describes the structure of a particular group of labels. Formally, a signature $\sigma$ is a tuple $(\text{params}, \text{typ})$ where $\text{params} \in \mathcal{P}(\text{Var})$ are variables representing label parameters, and $\text{typ} : \text{Var} \to \text{Typ}$ is a function that maps each variable to its type.

---

$[4]$However, our synthesis algorithm is not tied to a particular grammar, and developers may construct their own grammar based on the insights from the problem domain.
Each label $a \in L$ is associated with exactly one signature (denoted $\text{sig}(a)$). For instance, given label $a = \text{encWrite(}$secret, keyX$)$, $\text{sig}(a) = \sigma_{\text{encWrite}}$ is the signature that describes the structure of all $\text{encWrite}$ labels. In particular, here $\text{params} = \{m, k\}$ and $\text{typ} = \{(m, \text{Msg}), (k, \text{Key})\}$, where $m$ and $k$ are variables that represent the message and key associated with each $\text{encWrite}$ label. Furthermore, every label $a \in L$ assigns a concrete value to each parameter that appears in its signature. For example, given label $a = \text{encWrite(}$secret, keyX$)$, $a.m = \text{secret}$, and $a.k = \text{keyX}$, where secret and keyX are constant values that represent a secret message and a particular key, respectively.

Separate expressions may be used to describe relationships between labels of different signatures. Given a pair of labels $a$ and $b$, let $\sigma_a$ and $\sigma_b$ be their respective signatures, and let $E(\sigma_a, \sigma_b)$ be an expression that describes a relationship between parameters of these signatures. Then, a mapping constraint $C$ can be defined in terms of expressions as follows:

$$C = \{m : L \rightarrow L \mid \forall a \in L : (\exists b \in L : E(\sigma_a, \sigma_b)(a, b)) \Rightarrow E(\sigma_a, \sigma_b)(a, m(a)) \land 
\not\exists b \in L : E(\sigma_a, \sigma_b)(a, b)) \Rightarrow a \not\in \text{dom}(m)\}$$

That is, if expression $E(\sigma_a, \sigma_b)$ evaluates to true over some pair $(a, b)$, then each $m \in C$ must be defined on $a$, and the pair $(a, m(a))$ must also satisfy $E$; otherwise, $m$ cannot be defined over $a$ (i.e., the expression does not allow $a$ to be mapped to any other label).

**Example.** In our running example, the signatures associated with event labels are as follows:

writeBob$(m : \text{Msg})$  writeEve$(m : \text{Msg})$  encWrite$(m : \text{Msg}, k : \text{Key})$

Separate constraint expressions are used to capture the mapping between writeBob and encWrite, and writeEve and encWrite; we will call these expressions $E_{\text{writeBob}}$ and $E_{\text{writeEve}}$, respectively. One possible mapping constraint that preserves the secrecy property may be expressed as follows:

$$E_{\text{writeBob}}(a, b) \equiv a.m = b.m \land (b.m = \text{secret} \Rightarrow b.k = \text{keyX})$$

$$E_{\text{writeEve}}(a, b) \equiv a.m = b.m \land (b.m = \text{secret} \Rightarrow b.k = \text{keyX})$$

Informally, these expressions stipulate that (1) the messages associated with the high-level and low-level events are identical, and (2) if the message being transmitted is a secret, the key associated with Bob must be used. Note that this constraint does not specify which keys should be used to encode public messages, since this decision is irrelevant to the secrecy property. This under-specification is desirable, since it gives more implementation freedom to the developer while preserving the desired property at the same time.

### 5.2 Algorithm

In this section, we describe an algorithm that produces a valid, locally optimal constraint as a solution to Problem 5 in Section 4.

To ensure that the algorithm terminates, we assume that the set of expressions that may be constructed using a given grammar is restricted to a finite set. This restriction can be enforced either by bounding the size of expressions, or finitizing the domains of data types (Msg and Key in our example). The specifics of the restriction are provided by the user as an input.

Our algorithm also assumes the existence of a verifier that is capable of checking whether a mapping satisfies a given property $S$. The verifier is used to implement function $\text{verify}(C, P, Q, S)$, which returns $\text{OK}$ if and only if every mapping allowed by constraint $C$ satisfies $S$, and $\text{NotOK}$ otherwise. In Section 6.1, we describe one instantiation of this synthesis algorithm using the Alloy Analyzer, a formal modeling and analysis tool [29].

**Naive algorithm** Given that the number of candidate expressions is finite, one simple algorithm involves simply enumerating and verifying the candidates one-by-one. This algorithm, although simple to implement, is likely to suffer from scalability issues for problems with a large number of candidates. Indeed, we
attempted to apply this algorithm to our OAuth case study (described later in Section 6), and discovered that enumerating the entire search space under a reasonable amount of time is not feasible.

**Generalization algorithm** We present an improved algorithm that takes a generalization-based approach to dynamically identify and prune undesirable parts of the search space. The intuition behind the algorithm is shown graphically in Figure 2. A key observation is that only a few implementation decisions—captured by some minimal subset of the entries in a mapping—may be sufficient to imply that the resulting implementation will be invalid. Thus, given some invalid constraint $C$, the algorithm attempts to identify this minimal subset and construct a larger constraint $C_{bad}$ that is guaranteed to contain only invalid mappings.

The outline of the algorithm is shown in Algorithm 1. The function `synthesize` takes four inputs: a pair of processes, $P$ and $Q$, a desired specification $S$, and a user-specified constraint $C_0$. It also stores a set of constraints, $X$, which keeps track of “bad” regions of the search space that do not contain any valid mappings.

In each iteration, a constraint $C$ is selected from $\text{candidates}(C_0)$ (line 3), which represents the set of all possible non-empty candidates that satisfy the given constraint $C_0$ (i.e., for every $C$, $C \subseteq C_0$). If $C$ is subsumed by one of the constraints in $X$ (meaning, it is guaranteed to yield no valid solution), then it is simply discarded and a different candidate is selected (lines 4-5).

Given a candidate $C$, the verifier is used to check whether $C$ is valid with respect to $S$ (line 7). If so, then `generalize` is invoked on $C$ to produce a locally optimal mapping $C_{optimal}$, which represents the largest set that contains $C$ and is valid with respect to $S$ (line 9).

If, on the other hand, $C$ is invalid (meaning it allows at least one mapping that fails to preserve $S$), then `generalize` is invoked on $C$ to compute the largest superset of $C$ that contains only invalid mappings (i.e., those that satisfy $\neg S$). The set $C_{bad}$ is then added to $X$ and used to prune out subsequent, invalid candidates (line 13).

**Constraint generalization** The function `generalize($C$, $P$, $Q$, $S$, $C_0$)` computes the largest set that contains $C$ and only permits mappings that satisfy $S$. This function is used both to identify an undesirable region of the candidate space that should be avoided, and also produce a locally optimal version of a valid mapping constraint.

The procedure works by incrementally growing $C$ into a larger set $C'$ and stopping when $C'$ contains at least one mapping that violates $S$. Suppose that constraint $C$ is represented by a symbolic expression $E$, which itself is a conjunction of $n$ subexpressions $k_1 \land k_2 \land ... \land k_n$, where each $k_i$ for $1 \leq i \leq n$ represents a (possibly conditional) assignment of a variable or a constant to some label parameter. The function `decompose($C$)` takes the given constraint and returns the set of such subexpressions. The function `relax($C$, $k_i$)` then computes a new constraint by removing $k$ from $C$; this new constraint, $C'$, is a larger set of mappings that subsumes $C$.

The verifier is then used to check $C'$ against $S$ (line 22). If $C'$ is still valid with respect to $S$, it implies that the implementation decision encoded by $k$ is irrelevant to $S$, meaning we can safely remove $k$ from the final synthesized constraint $C$ (line 24). If not, $k$ is retained as part of $C$, and the algorithm moves onto the next subexpression $k$ as a candidate for removal (line 20).
fun synthesize(P, Q, S, C₀)

X = {}

for C ∈ candidates(C₀) do

if ∃ Cᵦ ∈ X : C ⊆ Cᵦ then
    skip

end

result ← verify(C, P, Q, S)

if result = OK then

Coptimal ← generalize(C, P, Q, S, C₀)

return Coptimal

else

Cᵦ ← generalize(C, P, Q, ¬S, C₀)

X ← X ∪ {Cᵦ}

end

end

return none

end

fun generalize(C, P, Q, S, C₀)

K ← decompose(C)

for k ∈ K do

C' ← relax(C, k)

result ← verify(C', P, Q, S)

if result = OK ∧ C' ⊆ C₀ then

C ← C'

end

end

return C

end

Algorithm 1: An algorithm for synthesizing a valid, locally optimal mapping constraint.

Example. Back to our running example, one possible candidate constraint C (line 3) is represented in part by the following expression (for mappings from writeEve to encWrite):

\[ E_{\text{writeEve}}(a, b) \equiv a.m = b.m \land (b.m = \text{secret} \Rightarrow b.k = \text{key}_X) \land (b.m = \text{public} \Rightarrow b.k = \text{key}_Y) \]

The verifier returns OK after checking C against P, Q, and S (line 7), meaning C is a valid constraint.

In the next step, the generalization procedure removes the subexpression \( k₁ = (b.m = \text{public} \Rightarrow b.k = \text{key}_Y) \) from C, resulting in new constraint \( C' \) that is represented as:

\[ E'_{\text{writeEve}}(a, b) \equiv a.m = b.m \land (b.m = \text{secret} \Rightarrow b.k = \text{key}_X) \]

When checked by the verifier (line 22), \( C' \) is still considered valid, meaning that the decision encoded by \( k₁ \) is irrelevant to the property \( \neg\Diamond (\text{SecretLearned}) \); thus, \( k₁ \) can be safely removed.

However, removing \( k₂ = (b.m = \text{secret} \Rightarrow b.k = \text{key}_X) \) leads to a property violation (causing the verifier to return NotOK). Thus, \( k₂ \) is kept as part of the final, locally optimal constraint expression.

6 Case Study

This section describes an implementation of our synthesis technique, and a case study on applying this technique to security protocols. In particular, the goal of the study is to answer (1) whether our technique can be used to synthesize valid implementation mappings for realistic systems, and (2) how effective our
generalization-based algorithm is over the naive approach.

6.1 Implementation

We have built a prototype tool that is capable of performing both the naive and generalization algorithms described in Section 5.2. Our tool is currently plugged in with two different verifiers: Spin [24], an explicit-state model checker, and the Alloy Analyzer [29], a modeling and analysis tool based on a first-order relational logic. Both tools have their strengths and weaknesses [47]. Spin provides stronger guarantees in that it is capable of fully exploring the state space, whereas Alloy performs a kind of bounded model checking (where the maximum length of traces explored is restricted to some fixed bound). On the other hand, in our experience so far, Alloy tends to find counterexamples more quickly (if they exist within the given bound), in part thanks to its constraint solving backend.

We began with an implementation that employed Spin as a verifier. Through our experiments, we discovered that as the size of the relaxed constraint $C'$ increased during the iterative generalization procedure (line 21 in Algorithm 1), the verification task by Spin became more demanding, eventually becoming a major bottleneck in the synthesis algorithm. We then employed the Alloy Analyzer as the verifier, and found that it did not suffer from the same issue. We believe that this is partly due to the constraint-based nature of Alloy: The relaxation step involves simply removing a constraint from the Alloy model, and does not adversely affect the performance of the constraint solver (in some cases, it leads to improvement).

6.2 OAuth Protocols

As a major case study, we took on the problem of synthesizing valid mappings for OAuth, a popular family of protocols used to carry out a process called third-party authorization [28]. The purpose of OAuth is to allow an application (called a client in the OAuth terminology) to access a resource from another application (an authorization server) without needing the credentials of the resource owner (an user). For example, a gaming application may initiate an OAuth process to obtain a list of friends from a particular user’s Facebook account, provided that the user has authorized Facebook to release this resource to the client.

In particular, we chose to study two versions of OAuth—OAuth 1.0 and 2.0. Although OAuth 2.0 is intended to be a replacement for OAuth 1.0, there has been much contention within the developer community about whether it actually improves over its predecessor in terms of security. For this reason, certain major websites (such as Twitter and Flickr) still rely on OAuth 1.0, while others have adopted 2.0. In fact, the original creator of OAuth himself has recommended 1.0 as the more secure version [19]:

...OAuth 2.0 at the hand of a developer with deep understanding of web security will likely result is a secure implementation. However, at the hands of most developers...2.0 is likely to produce insecure implementations.

Since both protocols are designed to provide the same security guarantees (i.e., both share a common property $S$), our goal was to apply our synthesis approach to systematically compare what developers would be required to do in order to construct secure web-based implementations of the two. We began by constructing and verifying abstract protocol models against $S$ to confirm that they indeed provide the necessary security guarantees at the abstract level. We then applied our technique to synthesize valid mappings from the two protocols to a model of the HTTP platform. The rest of this section describes our experimental procedure and results.

6.3 Experimental Setup

OAuth models We constructed models of OAuth 1.0 and 2.0 based on the official protocol specifications [28, 27] in Alloy. Due to limited space, we give only a brief overview of the models. Each model consists of four processes: Client, AuthServer, and two user processes, Alice and Eve, the latter with a malicious intent to access Alice’s resources.
Figure 3: A high-level overview of the two OAuth protocols, with a sequence of event labels that describe protocol steps in the typical order that they occur. Each arrowed edge indicates the direction of the communication. Variables inside labels with the prefix ret_ represent return parameters. For example, in Step 2 of OAuth 2.0, User passes her user ID and password as arguments to AuthServer, which returns ret_code back to User in response.

Let us first describe the simpler of the two protocols, OAuth 2.0. A typical 2.0 workflow, shown in Figure 3(a), begins with a user (Alice or Eve) initiating a new protocol session with Client (initiate). The user is then asked to prove her own identity to AuthServer (by providing a user ID and a password) and officially authorize the client to access her resources (authorize). Given the user’s authorization, the server then allocates a unique code for the user, and then redirects her back to the client. The user forwards the code to the client (forward), which then can exchange the code for an access token to her resources (getToken).

Like in OAuth 2.0, a typical workflow in 1.0 (depicted in Figure 3(b)) begins with a user initiating a new session with Client (initiate). Instead of immediately directing the user to AuthServer, however, Client first obtains a request token from AuthServer and associates it with the current session (getReqToken). The user is then asked to present the same request token to AuthServer and authorize Client to access her resources (authorize). Once notified by the user that the authorization step has taken place (notify), Client exchanges the request token for an access token that can be used subsequently to access her resources (getAccessToken).

We specified two desirable properties of OAuth: (1) progress: Each protocol session can be completed with the client obtaining an access token for a user, and (2) integrity: When the client receives an access token, it must correspond to the user who initiated the current protocol session. To ensure the validity of the OAuth models before the synthesis step, we performed verification of the properties using both Alloy and Spin; both properties were successfully verified in several seconds.

HTTP platform model In this case study, our goal was to explore and synthesize web-based implementations of OAuth. For this purpose, we constructed a model depicting interaction between a generic HTTP server and web browser. To ensure the fidelity of our model, we studied, as references, similar efforts by other researchers in building reusable models of the web for security analysis [2, 7, 10] (none of these models, however, has been used for synthesis).

The model contains two types of processes, Server and Browser (which may be instantiated into multiple processes representing different servers and browsers). They interact with each other over a sequence of HTTP requests, which share the following signature:

```
req(method : Method, url : URL,
      headers : List[Header], body : Body, ret_resp : Resp)
```

The parameters of an HTTP request have their own internal structures, each consisting of its own parameters.
Figure 4: Partial mapping specification from OAuth 2.0 to HTTP labels. Terms highlighted in blue and red are variables that represent the parameters inside OAuth and HTTP labels, respectively. For example, in forward, the abstract parameters code and session may be transmitted as part of an URL query, a header, or the request body, although its URL is fixed to http://client.com/forward.

Our model describes generic, application-independent HTTP interactions. In particular, each Browser process is a machine that constructs, at each communication step with Server, an arbitrary HTTP request by non-deterministically selecting a value for each parameter of the request; Server generates the corresponding response in a similar, non-deterministic manner. The processes, however, follow a platform-specific logic; for instance, when given a response from Server that instructs a browser cookie to be stored at a particular URL, Browser will include this cookie along with every subsequent request directed at that URL.

In addition, the model includes a process that depicts the behavior of a web attacker, who may operate her own site with malicious pages (to be visited by a browser user), or exploit weaknesses in a browser to manipulate the user into sending certain HTTP requests.

Mapping from OAuth to HTTP Building a web-based implementation of OAuth involves decisions about how abstract protocol operations are to be realized in terms of HTTP requests. As an input to the synthesizer, we specified an initial set of constraints that describe partial implementation decisions; the ones for OAuth 2.0 are shown in Figure 4. These decisions include a designation of fixed host and path names inside URLs for various OAuth operations (e.g., http://client.com/initiate for the OAuth initiate event), and how certain parameters are transmitted as part of an HTTP request (ret_session as a return cookie in initiate). We believe that it is reasonable to treat these constraints as given, since they describe decisions that are common across typical web-based OAuth implementations.

6.4 Results Our synthesis tool was able to generate valid, locally optimal mapping constraints for OAuth 2.0 and 1.0. In particular, the constraints describe mitigations against attacks that exploit an interaction between the OAuth logic and a browser vulnerability, including session swapping [42], covert redirect [18] (both for OAuth 2.0), and session fixation [17] (for OAuth 1.0). As an example, let us describe how certain implementation decisions may lead to one of these attacks, and how our synthesized mappings mitigate against this attack.
Insecure mapping Consider OAuth 2.0 from Figure 3(a). In order to implement the forward operation, for example, the developer must determine how the parameters of the abstract event label (code and session) are encoded using their concrete counterparts in an HTTP request. A number of choices is available. In one possible implementation, the authorization code may be transmitted as a query parameter inside the URL, and the session as a browser cookie, as described by the following constraint expression, $E_1$:

$$E_1(a, b) \equiv (b \text{. method} = \text{POST}) \land (b \text{. url} \text{. host} = \text{client.com}) \land$$
$$\quad (b \text{. url} \text{. path} = \text{forward}) \land (b \text{. url} \text{. queries}[0] = a \text{. code}) \land$$
$$\quad b \text{. headers}[0].\text{name} = \text{cookie} \land b \text{. headers}[0].\text{value} = a \text{. session}$$

where POST, client.com, forward, and cookie belong to are predefined constants; the $l[i]$ refers to $i$-th element of list $l$.

This constraint, however, permits a vulnerable implementation that fails to satisfy the integrity property for OAuth. In this attack, malicious user Eve performs the first two steps of the workflow in Figure 3(a) using her own credentials, and obtains a unique code ($\text{code}_{Eve}$) from the authorization server. Instead of forwarding this to Client (as she is expected to), Eve keeps the code herself, and crafts her own web page that would trigger the visiting browser to send the following HTTP request:

```plaintext
req(POST, http://client.com/forward?code_{Eve}; …)
```

Suppose that Alice is a naive browser user who may occasionally be enticed or tricked into visiting malicious web sites. When Alice visits the page set up by Eve, Alice’s browser automatically generates the above HTTP request, which, given the decisions in $E_1$, corresponds to a valid forward event:

```plaintext
forward(code_{Eve}, session_{Alice}) \mapsto
req(POST, http://client.com/forward?code_{Eve}, [(cookie, session_{Alice})], …)
```

Due to the standard browser behavior, the cookie corresponding to session_{Alice} is included in every request to client.com. As a result, Client will mistakenly accept code_{Eve} as the one for Alice, even though it actually belongs to Eve; this leads to a violation of the integrity property for OAuth.

Synthesized, secure mapping A major contributing factor to the above attack is that in a browser-based implementation, the client cannot trust an authorization code as having originated from a particular user (e.g., Alice), since the code may be intercepted or interjected by an attacker (Eve) while in transit through a browser. A possible solution is to explicitly identify the origin of the code by requiring an additional piece of tracking information to be provided in each forward request. The following mapping constraint $E_2$, synthesized by our tool, encodes one form of this solution:

$$E_2(a, b) \equiv (b \text{. method} = \text{POST}) \land (b \text{. url} \text{. host} = \text{client.com}) \land$$
$$\quad (b \text{. url} \text{. path} = \text{forward}) \land (b \text{. url} \text{. queries}[0] = a \text{. code}) \land$$
$$\quad b \text{. headers}[0].\text{name} = \text{cookie} \land b \text{. headers}[0].\text{value} = a \text{. session}$$
$$\quad (a \text{. session} = \text{session}_{\text{Alice}} \Rightarrow b \text{. url} \text{. queries}[1] = \text{nonce}_{a}) \land$$
$$\quad (a \text{. session} = \text{session}_{\text{Eve}} \Rightarrow b \text{. url} \text{. queries}[1] = \text{nonce}_{b})$$

where nonce_{a}, nonce_{b} ∈ Nonce are constants defined in the HTTP model. In particular, $E_2$ stipulates that every forward request must include an additional value (nonce) as an argument besides the code and the session, and that this nonce be unique for each session value. This mapping constraint $E_2$ ensures that the resulting implementation satisfies the properties for OAuth.

Although the above attack and possible mitigations have been informally discussed within the security community (referred as session swapping in [32]), our study is the first to automatically synthesize implementation decisions that mitigate against the attack.

---

6A nonce is a unique piece of string intended to be used once in communication.
### Table 1: Experimental Results

| Protocol  | # total candidates | # explored | # verified | # skipped | Avg. verif. time |
|-----------|--------------------|------------|------------|-----------|------------------|
| OAuth 1.0 | 79200              | 2465       | 281        | 2184      | 2.01             |
| OAuth 2.0 | 29400              | 1453       | 161        | 1292      | 1.88             |

| Protocol  | Verif. time | Gen. time | Total time | Naive alg. time | Speed up (x) |
|-----------|-------------|-----------|------------|-----------------|--------------|
| OAuth 1.0 | 566.05      | 490.84    | 1056.89    | 4732.63         | 4.48         |
| OAuth 2.0 | 302.76      | 1138.85   | 1441.60    | 2717.41         | 1.88         |

Figure 5: Experimental results on the OAuth mapping synthesis. All times are in seconds. “Verif. time” and “Gen. time” are the total amounts of time spent on verification and generalization, respectively. “Total time” refers to the overall time spent by the generalization-base algorithm to synthesize an optimal constraint.

**OAuth 1.0 vs 2.0**

Based on the comparison of the two protocol workflows in Figure 3, OAuth 2.0 appears simpler than its predecessor, which requires the client to perform an extra step to obtain a request token from the authorization server (Step 2 in Figure 3(b)).

Simplicity, however, sometimes can result in a loss of security. In OAuth 1.0, the client knows exactly the request token that it expects to receive in Step 4 (the same one as in Step 2). Thus, it needs not trust that the user will always deliver a correct request token in Step 4, and does not suffer from the above attack. OAuth 2.0, on the other hand, relies on the user to deliver a correct authorization code (Step 3 in Figure 3(a))—an assumption which does not necessarily hold in a browser-based implementation, as described above. In effect, the synthesized mapping $E_2$ captures a way to harden OAuth 2.0 implementations against security risks of relying on this assumption.

**Performance**

Figure 5 shows statistics from our experiments synthesizing locally optimal, valid mapping constraints for the two OAuth protocols. Overall, the synthesizer took approximately 17.6 and 24.0 minutes to synthesize the constraints for 1.0 and 2.0, respectively. We believe that this level of performance is acceptable for generic protocols such as OAuth: Once synthesized, the implementation guidelines captured by the constraints can be consulted by multiple developers to build their own OAuth implementations, amortizing the cost of the synthesis effort over time.

In both cases, the tool spent a considerable amount of time on the generalization step to learn the invalid regions of the search space. It can also be seen that the generalization is effective at identifying and discarding a large number of invalid candidates, and achieves a significant amount of speed up over the naive algorithm (4.48 and 1.88 times for OAuth 1.0 and 2.0, respectively).

Since OAuth 1.0 is a more complex protocol than 2.0, we expected the synthesis procedure to take more time on the former; indeed, the performance of the naive algorithm was consistent with this expectation (4732.63 vs 2717.41 seconds). However, we were surprised to find that the generalization-based algorithm was able to find a solution for OAuth 1.0 faster than it did for 2.0. In particular, the synthesizer spent a significantly larger proportion of its time on the generalization step (1138.85/1441.60 ≈ 79%); one possible explanation is that the search space for OAuth 2.0 is more densely populated with incomparable invalid constraints than 1.0.

As an additional experiment, we ran the generalization-based algorithm to explore the search space exhaustively without terminating when an optimal solution is found. The algorithm was able to skip 75375 out of 79200 candidates for OAuth 1.0 (roughly 95%), and 28140 out of 29400 candidates for OAuth 2.0 (≈ 96%). This means that only 3825 and 1260 calls to the verifier were needed to exhaust the search space (for 1.0 and 2.0, respectively)—relatively small numbers compared to the total number of candidates.

---

7The experiments were performed on a Mac OS X 2.7 GHz machine with 8G RAM and MiniSat as the underlying SAT solver employed by the Alloy Analyzer.
6.5 Discussions

Lessons learned  The input mapping constraint (depicted in Figure 4), capturing the developer’s partial knowledge or decisions, was crucial for reducing the size of the search space and keeping the synthesis procedure tractable. We believe that in practice, our tool will be more effective as a completion tool—allowing the developer to express her partial knowledge using a constraint, and filling in gaps that represent her uncertainty—rather than one that generates an entire set of implementation decisions from scratch.

The generalization algorithm was particularly effective for the OAuth protocols, since a significant percentage of the candidate constraints would result in an implementation that violates the progress property (i.e., it prevents Alice or Eve from completing a protocol session in an expected order). Often, the decisions contributing to this violation could be localized to a small subset of entries in a mapping (for example, attempting to send a cookie to a mismatched URL, which is inconsistent with the behavior of the browser process). By identifying this subset, our algorithm was able to discover and eliminate a large number of invalid mappings.

Threats to validity  One potential source of errors in our study is the scope and accuracy of the models used to describe the OAuth protocols and HTTP platform. The security guarantees provided by synthesized mappings are limited to the extent which these models are accurately able to capture potential attacks on real OAuth implementations. For example, our synthesized mappings do not protect an implementation against attacks that exploit a vulnerability called cross-site scripting: Modeling this vulnerability would involve details about how individual HTML pages are sanitized, which, in turn, depend on the underlying web development framework or libraries used. However, we believe that this is not an inherent flaw in our synthesis technique, but rather a risk that may arise in any model-based approach to verification and synthesis.

Another potential source of errors stems from the finitization of the system models. To finitize the set of labels \( L \) and ensure that the synthesis procedure terminates, we bounded the size of datatype domains to 4 (i.e., 4 access tokens, cookies, etc.). While we believe that this bound is sufficient to explore possible interactions between protocol participants during an OAuth session, it is possible that we might have missed a potential security violation involving a larger number of data elements.

7 Related Work

A large body of literature exists on refinement-based methods to system construction [22, 6]. These approaches involve building an implementation \( Q \) that is a behavioral refinement of \( P \); such \( Q \), by construction, would satisfy the properties of \( P \). In comparison, we start with an assumption that \( Q \) is a given platform, and that the developer may not have the luxury of being able to modify or build \( Q \) from scratch. Thus, instead of behavioral refinement (which may be too challenging to achieve), we aim to preserve some critical property \( S \) when \( P \) is implemented using \( Q \).

Our algorithm can be considered as a kind of counterexample-driven inductive synthesis (CEGIS) [40], in which the verifier is used as an oracle to learn negative regions of the search space and guide the synthesizer towards a desirable solution.

The task of synthesizing a valid mapping can be seen as a type of the model merging problem [10]. This problem has been studied in various contexts, including architectural views [32], behavioral models [8, 44], and database schemas [37]. Among these, our work is most closely related to merging of partial behavioral models [8, 44]. In these works, given a pair of models \( M_1 \) and \( M_2 \), the goal is to construct \( M' \) that is a behavioral refinement of both \( M_1 \) and \( M_2 \). The approach proposed in this paper differs in that (1) the mapping composition involves merging a pair of events with distinct alphabet labels into a single event that retains all of those labels, and (2) the composed process \( (P \parallel_m Q) \) needs not be a behavioral refinement of \( P \) or \( Q \), as long as it satisfies property \( S \).

Full abstraction is a notion developed to reason about equivalence between the operational and denotational semantics of a programming language [33, 36]. This notion has been adopted in the context of security,
to reason about issues that arise when a program in one language is translated to a (typically, lower-level) representation in another language [1]. A property violation that arises due to an invalid mapping can be seen as a kind of full abstraction violation. However, relatively little work has been done exploring synthesis methods for achieving full abstraction; we believe that our work is a step towards this goal.

Bhargavan and his colleagues present a compiler that takes a high-level program written using session types [25] and automatically generates a low-level implementation [9]. This technique is closer to compilation than to synthesis in that it uses a fixed translation scheme from high-level to low-level operations in a specific language environment (.NET), without searching a space of possible translations.

Specware is a deductive synthesis framework for deriving an implementation from a high-level specification through a series of refinement steps [26]. Specware is more general in that their goal is to provide a general-purpose development environment, whereas mappings described in this paper target one kind of implementation decisions. On the other hand, Specware requires a considerable amount of manual guidance from the user during the refinement steps, while our goal is to enable fully automated synthesis.

Our approach is similar to a number of other synthesis frameworks [5, 30, 39, 41] in allowing the user to provide a partial specification of the artifact to be synthesized (often in form of constraints or examples), thus having the underlying engine complete the remaining parts. This is especially crucial for pruning the space of candidate solutions and keeping the synthesis problem tractable. Synthesizing a low-level implementation from a high-level specification has also been studied in the context of data structure synthesis [14, 20, 21].

8 Conclusions and Future Work

Currently, the notion of mapping is one-to-one between labels; that is, each mapping relates each label to at most one other label. This notion, however, may be not expressive enough to capture certain kinds of implementation decisions that involve realizing one abstract label as a series of multiple, concrete labels. To be able to model and synthesize these decisions, the framework needs to be extended to support a different kind of mapping—one that maps a label to a sequence of other labels.

As with many synthesis problems, scalability remains a challenge. One promising direction is to exploit the fact that our generalization-based algorithm (from Section 5.2) is easily parallelizable. We are currently devising an algorithm where multiple machines are used to explore different regions of the search space in parallel, only communicating to update and check the invalid constraint set \( X \).

Our synthesis tool can be used in an interactive manner. If it fails to find a valid mapping constraint due to a given constraint \( C \) that is too restrictive, the developer may relax \( C \) and re-run the synthesis procedure. However, the tool currently does not provide an explanation for why it fails to synthesize a mapping; such an explanation could point to parts of \( C \) that must be relaxed, or certain behavior of \( Q \) that entirely precludes valid mappings. We are developing a root cause analysis technique that can be used to provide such explanations.

Another major next step is to bridge the gap between synthesized mappings (which describe implementation decisions at a modeling level) and code. For instance, a mapping may be used to directly generate a working implementation that preserves a desired property (e.g., a secure, reference OAuth implementation), or used as a code-level specification to check that a program adheres to the decisions described in the mapping. We are also exploring potential applications of our synthesis approach to other application domains where a similar type of mapping arises, such as cyber-physical and embedded systems [11, 38, 43].

Acknowledgements

This work was partially supported by the NSF Breakthrough award “Compositional System Modeling” (CNS-132975), the NSF Expeditions in Computing project “ExCAPE: Expeditions in Computer Augmented Program Engineering” (CCF-1138860 and 1139138), and by the Academy of Finland.
References

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

[2] 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.

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. Inf. Process. Lett., 21(4):181–185, 1985.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pages 1–8, 2013.

[5] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic Completion of Distributed Protocols with Symmetry. In 27th International Conference on Computer Aided Verification (CAV), volume 9207 of LNCS, pages 395–412. Springer, 2015.

[6] Ralph-Johan Back. A calculus of refinements for program derivations. Acta Inf., 25(6):593–624, 1988.

[7] 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.

[8] 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.

[9] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Denièlou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009, pages 124–140, 2009.

[10] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In Proceedings of the 2006 international workshop on Global Integrated Model Management, pages 5–12, 2006.

[11] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications. In Proceedings of the 2003 ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES’03), pages 153–162. ACM, June 2003.

[12] Suresh Chari, Charanjit Jutla, and Arnab Roy. Universally Composable Security Analysis of OAuth v2.0. IACR Cryptology ePrint Archive, 2011:526, 2011.

[13] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014, pages 892–903, 2014.

[14] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 689–700, 2015.
[15] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, pages 502–518, 2003.

[16] 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.

[17] OAuth Working Group. OAuth Security Advisory: 2009.1 “Session Fixation”. https://oauth.net/advisories/2009-1 2009.

[18] OAuth Working Group. OAuth Security Advisory: 2014.1 “Covert Redirect”. https://oauth.net/advisories/2014-1-covert-redirect 2014.

[19] Eran Hanmer. OAuth 2.0 and the Road to Hell. https://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell 2012.

[20] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pages 38–49, 2011.

[21] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12, Beijing, China - June 11 - 16, 2012, pages 417–428, 2012.

[22] C. A. R. Hoare. Proof of correctness of data representations. Acta Inf., 1:271–281, 1972.

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

[24] Gerard J. Holzmann. The SPIN Model Checker - primer and reference manual. Addison-Wesley, 2004.

[25] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, pages 273–284, 2008.

[26] Kestrel Institute. Specware. http://www.specware.org

[27] Internet Engineering Task Force. The OAuth 1.0 Protocol. https://tools.ietf.org/html/rfc5849, 2010.

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

[29] Daniel Jackson. Software Abstractions: Logic, language, and analysis. MIT Press, 2006.

[30] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pages 215–224, 2010.

[31] Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. Multi-Representational Security Analysis. In International Symposium on the Foundations of Software Engineering (FSE), 2016.

[32] 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.

[33] Robin Milner. Fully abstract models of typed lambda-calculi. Theor. Comput. Sci., 4(1):1–22, 1977.

[34] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve M. Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In ICSE, pages 54–64, 2007.
[35] 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.

[36] Gordon D. Plotkin. LCF considered as a programming language. Theor. Comput. Sci., 5(3):223–255, 1977.

[37] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In VLDB, pages 826–873, 2003.

[38] A. Sangiovanni-Vincentelli. Defining platform-based design. EE Times, February 2002.

[39] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pages 289–299, 2011.

[40] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit A. Seshia, and Vijay 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.

[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit A. Seshia, and Vijay 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.

[42] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In the ACM Conference on Computer and Communications Security, CCS’12, Raleigh, NC, USA, October 16-18, 2012, pages 378–390, 2012.

[43] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing Synchronous Models on Loosely Time-Triggered Architectures. IEEE Transactions on Computers, 57(10):1300–1314, October 2008.

[44] Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In SIGSOFT FSE, pages 43–52, 2004.

[45] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA, pages 365–379, 2012.

[46] 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.

[47] Pamela Zave. A practical comparison of alloy and spin. Formal Asp. Comput., 27(2):239–253, 2015.