Automated Synthesis of Secure Platform Mappings

Eunsuk Kang[1], Stéphane Lafortune[2], and Stavros Tripakis[3]

[1] Carnegie Mellon University eskang@cmu.edu
[2] University of Michigan stephane@umich.edu
[3] Northeastern University stavros@northeastern.edu

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 platform mapping*: synthesize a set of implementation decisions ensuring that a desired property is preserved from a high-level design into a low-level platform implementation. We formalize this synthesis problem and propose a technique for generating a mapping based on symbolic constraint search. We describe our prototype implementation, and two real-world case studies demonstrating the applicability of our technique to the synthesis of secure mappings for the popular web authorization protocols OAuth 1.0 and 2.0.

1 Introduction

When building a complex software system, one may begin by coming up with an abstract design, and then construct 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 its features and security vulnerabilities. As a result, achieving an implementation that perfectly conforms to the design—in the traditional notion of behavioral refinement [20]—may be too difficult in practice. Worse, the resulting implementation may not necessarily preserve desirable properties that have already been established at the level of 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 [9, 33, 42], has been shown to be vulnerable to attacks when implemented on a web browser or a mobile device [39, 41, 10]. 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*, *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 platform mapping*: A set of implementation decisions ensuring that a desired property is preserved from a high-level design into a low-level platform implementation.

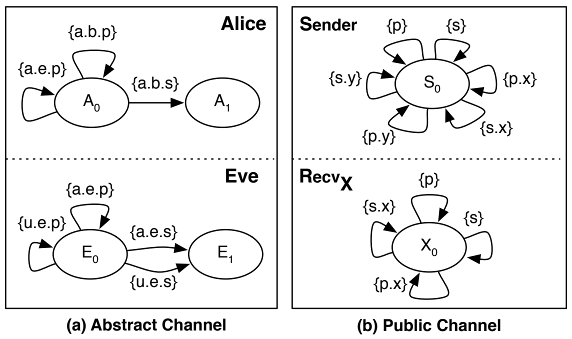
Our approach builds on the prior work of Kang et al. [28], which proposes a modeling and verification framework for reasoning about security attacks across multiple levels of abstraction. The central notion in this framework is that of a *mapping*, which captures a developer's decisions about how abstract system entities are to be realized in terms of their concrete counterparts. In this paper, we fix a bug in the formalization of mapping in [28] and extend the framework of [28] with the novel problem of synthesizing a property-preserving mapping. In addition, we present an algorithmic technique for performing this synthesis task. Our technique, inspired by the highly successful paradigms of *sketching* and *syntax-guided synthesis* [3, 38, 37, 26], takes a *constraint generalization* approach to (1) quickly prune the search space and (2) produce a solution that is *maximal* (i.e., a largest set of mappings that preserve a given property).

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 maximal set of mappings (if one exists) that would ensure that the resulting platform implementation preserves the given property. We have successfully applied our tool to synthesize property-preserving mappings for two non-trivial case studies: the authentication protocols OAuth 1.0 and 2.0 implemented on top of HTTP. Our 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 [39, 41].

The contributions of this paper include: a formal treatment of mapping, including a correction in the original definition [28] (Section 2); a formulation of the *mapping synthesis problem*, a novel approach for ensuring the preservation of a property between a high-level design and its platform implementation (Section 3); a technique for automatically synthesizing mappings based on symbolic constraint search (Section 4); and a prototype implementation of the synthesis technique along with a real-world case study demonstrating the feasibility of this approach (Section 5). We conclude with a discussion of related work (Section 6).

2 Mapping Composition

Our approach builds on the modeling and verification framework proposed by Kang et al. [28], which is designed to allow modular reasoning about behavior of processes across multiple abstraction layers. In this framework, a trace-based semantic model



L_{Alice} = { a.b.p, a.b.s, a.e.p, a.e.s }
L_{Eve} = { a.e.p, a.e.s, u.e.p, u.e.s }
$L_{\mathsf{Sender}} = L_{\mathsf{Recv_X}}$ =
{ p, s, p.x, s.x, p.y, s.y }

E_{Alice} = { {a.b.p}, {a.b.s}, {a.e.p} }
E_{Eve} = { {a.e.p}, {a.e.s}, {u.e.p}, {u.e.s} }
E_{Sender} = { {p}, {s}, {p.x}, {s.x}, {p.y}, {s.y} }
$E_{\mathsf{Recv_X}}$ = { {p}, {s}, {p.x}, {s.x} }

(c) Labels and events

Fig. 1. A pair of high-level (abstract) and low-level (public) communication models. Note that each event is a *set* of labels, where each label describes one possible representation of the event.

(based on CSP [21]) 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.

Running example. Consider a simple example involving communication of messages among a set of processes. In our modeling approach, the communication of a message is represented by labels of the form *sender.receiver.message*. For example, label a.e.p represents Alice sending Eve a public, non-secret message. Similarly, a.b.s represents Alice sending a secret message to another process (b for Bob, for example). In this system, Alice is unwilling to share its secret with Eve; in Figure 1(a), this is modeled by the absence of any transition on event {a.e.s} in the Alice process.

Eve is a malicious character whose goal is to learn Alice's secret. Beside a.e.p and a.e.s, Eve is associated with two additional labels, u.e.p and u.e.s, which represent receiving a public or secret message, respectively, through some *unknown* sender u. Conceptually, these two latter labels can be regarded as *side channels* [30] that Eve uses to obtain information.

A desirable property of this abstract communication system is that Eve should never be able to learn Alice's secret[4]. In this case, it can be easily observed that the property holds, since Alice, by design, never sends the secret to Eve.

The model in Figure 1(b) describes communication over a low-level public channel that is shared among all processes. A message sent over this channel may be encrypted using a key, as captured by labels of the form *message.key*. For instance, p.x and s.x represent the transmission of a public and secret message, respectively, using key x. A message may also be sent in plaintext by omitting an encryption key (e.g., label s represents the plaintext transmission of a secret). Each receiver on the public channel is assumed to have knowledge of only a single key; for instance, Recv$_X$ only knows key x and thus cannot receive messages that are encrypted using key y (i.e., labels p.y and s.y do not appear in events of Recv$_X$).

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

[4] A formalization of this property is provided later in this section.

particular, in the low-level implementation, Eve and other processes (e.g., Bob) are required to share the same channel, no longer benefitting from the separation provided by the abstraction in Figure 1(a). Does the property of the abstract communication hold in every possible implementation? If not, which decisions ensure that Alice's secret remains protected from Eve? We formulate these questions as the problem of synthesizing a *property-preserving mapping* between a pair of high-level and low-level models.

Events, traces, and processes. Let L be a potentially infinite set of labels. An *event* e is a finite, non-empty set of labels: $e \in E(L)$, where $E(L)$ is the set of all finite subsets of L except the empty set $\emptyset$. Let S^* be the set of all finite sequences of elements of set S. A *trace* t is a finite sequence of events: $t \in T(L)$, where $T(L)$ is the set of all traces over L (i.e., $T(L) = (E(L))^*$). The empty trace is denoted by $\langle\rangle$, and the trace consisting of a sequence of events $e_1, e_2, ...$ is denoted $\langle e_1, e_2, ...\rangle$. 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.

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 $t \upharpoonright A$, is defined as follows:

$$\langle\rangle \upharpoonright A = \langle\rangle \qquad (\langle e\rangle \cdot t) \upharpoonright A = \begin{cases} \langle e \cap A\rangle \cdot (t \upharpoonright A) & \text{if } e \cap A \neq \emptyset \\ (t \upharpoonright A) & \text{otherwise} \end{cases}$$

For example, if $t = \langle\{a\},\{a,c\},\{b\}\rangle$, then $t \upharpoonright \{a,b\} = \langle\{a\},\{a\},\{b\}\rangle$ and $t \upharpoonright \{b,c\} = \langle\{c\},\{b\}\rangle$.

A *process* P is defined as a triple (L_P, E_P, T_P). The *labels* of process P, $L_P \subseteq L$, is the set of all labels appearing in P, and $E_P \subseteq E(L)$ is the set of events that *may* appear in traces of P, which are denoted by $T_P \subseteq T(L)$. We assume traces in every process P to be *prefix-closed*; i.e., $\langle\rangle \in T_P$ and for every non-empty trace $t' = t \cdot \langle e\rangle \in T_P$, $t \in T_P$.

Parallel composition. A pair of processes P and Q synchronize with each other by performing events 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, let $P = (L_P, E_P, T_P)$ and $Q = (L_Q, E_Q, T_Q)$ be a pair of processes. Their parallel composition is defined as follows:

$$E_{P\|Q} = \{e \in E(L_P \cup L_Q) \mid \mathit{eventCond}(e,P) \wedge \mathit{eventCond}(e,Q) \wedge \mathit{syncCond}(e)\}$$
$$T_{P\|Q} = \{t \in (E_{P\|Q})^* \mid (t \upharpoonright L_P) \in T_P \wedge (t \upharpoonright L_Q) \in T_Q\} \qquad \textbf{(Def. 1)}$$

where $L_{P\|Q} = L_P \cup L_Q$, predicate *eventCond* is defined as

$$\mathit{eventCond}(e,P) \equiv e \cap L_P = \emptyset \vee e \cap L_P \in E_P$$

and a condition on synchronization, *syncCond*, is defined as

$$\mathit{syncCond}(e) \equiv e \subseteq L_P - L_Q \vee e \subseteq L_Q - L_P \vee (\exists a \in e : a \in L_P \cap L_Q) \quad \textbf{(Cond. 1)}$$

The definition of $T_{P\|Q}$ states 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 (**Cond. 1**) is imposed on every event appearing in $T_{P\|Q}$ to ensure that an event performed together by P and Q contains at least one common label shared by both processes.

This type of parallel composition can be seen as a generalization of the parallel composition of CSP [21], from single labels to *sets* of labels. That is, the CSP parallel composition is the special case of the composition of **Def. 1** where every event is a singleton (i.e., it contains exactly one label). Note that if event e contains exactly one label a, then a must belong to the alphabet of P or that of Q, which means $syncCond(e)$ always evaluates to true. The resulting expression in that case

$$T_{P\|Q} = \{t \in T(L_P \cup L_Q) \mid (t \upharpoonright L_P) \in T_P \wedge (t \upharpoonright L_Q) \in T_Q\}$$

is equivalent to the definition of parallel composition in CSP [21, Sec. 2.3.3].

Mapping composition. A *mapping m over set of labels L* is a partial function $m : L \to L$. Informally, $m(a) = b$ stipulates that every event that contains a as a label is to be assigned b as an additional label. We sometimes use the notations $a \mapsto_m b$ or $(a,b) \in m$ as alternatives to $m(a) = b$. When we write $m(a) = b$ we mean that $m(a)$ is defined and is equal to b. The *empty* mapping, denoted $m = \emptyset$, is the partial function $m : L \to L$ which is undefined for all $a \in L$.

Mapping composition allows a pair of processes to interact with each other over distinct labels. Formally, consider two processes $P = (L_P, E_P, T_P)$ and $Q = (L_Q, E_Q, T_Q)$, and let $L = L_P \cup L_Q$. Given mapping $m : L \to L$, the *mapping composition $P\|_m Q$* is defined as follows:

$$\begin{aligned} E_{P\|_m Q} &= \{e \in E(L_P \cup L_Q) \mid \ eventCond(e,P) \wedge eventCond(e,Q) \wedge \\ &\qquad\qquad\qquad\qquad syncCond'(e) \wedge mapCond(e,m)\} \\ T_{P\|_m Q} &= \{t \in (E_{P\|_m Q})^* \mid (t \upharpoonright L_P) \in T_P \wedge (t \upharpoonright L_Q) \in T_Q\} \qquad \textbf{(Def. 2)} \end{aligned}$$

where $L_{P\|_m Q} = L_P \cup L_Q$, and $syncCond'(e)$ and $mapCond(e,m)$ are defined as:

$$syncCond'(e) \equiv syncCond(e) \vee (\exists\, a \in e \cap L_P, \exists\, b \in e \cap L_Q : m(a) = b \vee m(b) = a)$$

$$mapCond(e,m) \equiv (\forall\, a \in e : a \in dom(m) \Rightarrow m(a) \in e)$$

where $dom(m)$ is the domain of function m. Compared to **Def. 1**, the additional disjunct in $syncCond'(e)$ 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 predicate $mapCond$ ensures that if an event e contains a label a and m is defined over a, then e also contains the label that a is mapped to.

Note that **Def. 2** is different from the definition of mapping composition in [28], and corrects a flaw in the latter. In particular, the definition in [28] omits condition $syncCond'$, which permits the undesirable case in which events e_1 and e_2 from P and Q are synchronized into union $e = e_1 \cup e_2$ even when the events do not share any label.

Example. Let P and Q be the abstract and public channel communication models from Figure 1(a) and (b), respectively. The property that Eve never learns Alice's secret can be stated as follows:

$$\Phi \equiv \neg(\exists\, e \in E(L) : l_1, l_2 \in e : l_1 = \mathsf{a.*.s} \wedge l_2 = \mathsf{*.e.s})$$

where $* \in \{a, b, e, u\}$. In other words, Eve should never be able to engage in an event that involves the transmission of Alice's secret. From Figure 1(a), it can be observed that $P = \mathsf{Alice}\|\mathsf{Eve} \models \Phi$.

Suppose that we decide on a simple implementation scheme where the abstract messages sent by Alice are transmitted over the public channel in plaintext; this decision can be encoded as a mapping, m_1, where each abstract label (i.e., L_{Alice} in Figure 1(c)) is mapped to concrete label p or s as follows:

$$\mathsf{a.b.p}, \mathsf{a.e.p}, \mathsf{u.e.p} \mapsto_{m_1} \mathsf{p} \qquad \mathsf{a.b.s}, \mathsf{a.e.s}, \mathsf{u.e.s} \mapsto_{m_1} \mathsf{s}$$

The resulting implementation can be constructed as process $I_{m_1} \equiv (\mathsf{Alice} \|_{m_1} \mathsf{Sender}) \parallel (\mathsf{Eve} \|_{m_1} \mathsf{Recv_X})$. Due to the definition of mapping composition (**Def. 2**), the following event may appear in a trace of the overall composite process:

$$\langle \{\mathsf{a.b.s}, \mathsf{s}, \mathsf{a.e.s}\} \rangle \in T_{I_{m_1}}$$

Note that this trace is a violation of the above property (i.e., $I_{m_1} \not\models \Phi$). This can be seen as an example of *abstraction violation*: As a result of decisions in m_1, a.b.s and u.e.s now share the same underlying representation (s), and Eve is able to engage in an event with a label (a.b.s) that was not previously available to it in the abstract model.

Properties of the mapping composition operator. Mapping composition is a generalization of parallel composition: The latter is a special case of mapping composition where the given mapping is empty:

Lemma 1. *Given a pair of processes P and Q, if $m = \emptyset$ then $P\|_m Q = P \parallel Q$.*

Commutativity. The proposed mapping composition operator is commutative: i.e., $P\|_m Q = Q\|_m P$. This property can be inferred from the fact that **Def. 2** is symmetric with respect to P and Q. It follows that by being a special case of mapping composition, the parallel composition operator is also commutative.

Associativity. The mapping composition operator is associative under the following conditions on the alphabets of involved processes and mappings:

Theorem 1. *Given processes P, Q, and R, let $X = (P\|_{m_1} Q)\|_{m_2} R$ and $Y = P\|_{m_3}(Q\|_{m_4} R)$. If $E_X = E_Y$, then $X = Y$.*

Proof. Available in the extended version of this paper [27].

3 Synthesis Problems

The *mapping verification problem* is to check, given processes P and Q, mapping m, and specification Φ, whether $(P\|_m Q) \models \Phi$. This problem was studied by Kang et al. [28]. In this paper, we introduce and study, for the first time to our knowledge, the problem of *mapping synthesis*. We begin with a simple formulation of the problem and then generalize it. We will not define what exactly the specification Φ may be, neither the satisfaction relation $\models$, as the mapping synthesis problems defined below are generic and can work with any type of specification or satisfaction relation. In Section 5.1, we discuss how this generic framework is instantiated in our implementation.

*Problem 1 (**Mapping Synthesis**).* Given processes P and Q, and specification Φ, find, if it exists, a mapping m such that $(P\|_m Q) \models \Phi$. We call such an m a *valid* mapping.

Note that if Φ is a *trace* property [2, 29], this problem can be stated as a $\exists\forall$ problem; that is, finding a witness m to the formula $\exists\, m : \forall\, t \in T_{P\|_m Q} : t \in \Phi$.

Instead of synthesizing m from scratch, the developer may wish to express their 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 2 (**Generalized Mapping Synthesis**).* Given processes P and Q, specification Φ, and mapping constraint C, find, if it exists, a valid mapping m such that $m \in C$.

Note that Problem 1 is a special case of Problem 2 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 3 (**Mapping Constraint Synthesis**).* Given processes P and Q, specification Φ, and mapping constraint C, generate, if it exists, a non-empty set of valid mappings C' such that $C' \subseteq C$. We call such a C' valid with respect to P, Q, Φ and C.

A procedure for solving Problem 3 can be used to solve Problem 2: Having generated constraint C', we can pick any mapping $m \in C'$. Such an m is guaranteed to be valid and also to belong in C.

In practice, it is desirable for C' to be as large as possible while still being valid, as it provides more implementation choices (i.e., possible mappings). In particular, we say that a mapping constraint C' is *maximal* with respect to P, Q, Φ, and C if and only if (1) C' is valid with respect to P, Q, Φ, and C, and (2) there exists no other constraint C'' such that C'' is also valid w.r.t. P, Q, Φ, C, and $C' \subseteq C''$. Then, our final synthesis problem can be stated as follows:

*Problem 4 (**Maximal Constraint Synthesis**).* Given processes P and Q, property Φ, and constraint C, generate, if it exists, a maximal constraint C' with respect to P, Q, Φ, C.

If found, C' is a *local* optimal solution. In general, there may be multiple maximal constraints for given P, Q, Φ, and C.

Example. Back to our running example, an alternative implementation of the abstract communication model over the public channel involves encrypting messages sent by Alice to Bob using a key (y) that Eve does not possess; this decision can be encoded as the following *valid* mapping m_2:

$$\mathsf{a.b.p} \mapsto_{m_2} \mathsf{p.y} \quad \mathsf{a.b.s} \mapsto_{m_2} \mathsf{s.y} \quad \mathsf{a.e.p} \mapsto_{m_2} \mathsf{p.x} \quad \mathsf{a.e.s} \mapsto_{m_2} \mathsf{s.y}$$

Since Eve cannot read messages encrypted using key y, she is unable to obtain Alice's secret over the public channel; thus, $I_{m_2} \models \Phi$, where $I_{m_2} \equiv (\mathsf{Alice} \|_{m_2} \mathsf{Sender}) \parallel (\mathsf{Eve} \|_{m_2} \mathsf{Recv}_{\mathsf{X}})$.

The following mapping, m_3, which leaves non-secret messages unencrypted in the low-level channel (as p), is also valid with respect to Φ:

$$\mathsf{a.b.p} \mapsto_{m_2} \mathsf{p} \quad \mathsf{a.b.s} \mapsto_{m_2} \mathsf{s.y} \quad \mathsf{a.e.p} \mapsto_{m_2} \mathsf{p} \quad \mathsf{a.e.s} \mapsto_{m_2} \mathsf{s.y}$$

since Eve being able to read non-secret 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 Alice's secret remains protected from Eve. In other words, $C_1 = \{m_2, m_3\}$ is a valid (but not necessarily maximal) 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.

4 Synthesis Technique

Mapping representation. In our approach, mappings are represented *symbolically* as logical expressions over variables that correspond to labels being mapped. The symbolic representation has the following advantages over an explicit one (where the entries of mapping m are enumerated explicitly): (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.

We adopt the symbolic representation and, inspired by SyGuS [3], use a *syntactic* approach where the space of candidate mapping constraints is restricted to expressions that can be constructed from a given grammar. Our grammar is specified as follows:

$$Term := Var \mid Const \qquad Assign := (Term = Term)$$
$$Expr := Assign \mid \neg Assign \mid Assign \Rightarrow Assign \mid Expr \land Expr$$

where Var is a set of variables that represent parameters inside a label, and $Const$ 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.

A mapping constraint is symbolically represented as a set of predicates, each of the form $\mathcal{X}(abs, conc)$ over *symbolic* labels abs and $conc$, where abs represents the label being mapped to $conc$. The body of each predicate is constructed as an expression from the above grammar. For example, let $abs = \mathsf{a}.\mathsf{b}.msg$ be a symbolic encoding of labels that represent Alice communicating to Eve, with variable msg corresponding to the message being sent; similarly, let $conc = msg'.key$ be a symbolic label in the public channel model, where msg' and key correspond to the message being transmitted and the key used to encrypt it (if any). Then, the expression

$$\mathcal{X}(\mathsf{a}.\mathsf{b}.msg, msg'.key) \equiv msg = msg' \land (msg = \mathsf{s} \Rightarrow key = \mathsf{y})$$

states that (1) parameter msg in the abstract label must be equal to that in the concrete label (i.e., the message being transmitted must be preserved during the mapping) and (2) if the message is a secret, key y must be used to encrypt it in the implementation.

The set of mappings that predicate $\mathcal{X}(abs, conc)$ represents is defined as:

$$C = \{m : L \to L \mid \forall\, abs \in L : (abs \in dom(m) \Leftrightarrow \exists\, conc \in L : \mathcal{X}(abs, conc)) \land$$
$$(abs \in dom(m) \Rightarrow \mathcal{X}(abs, m(abs)))\}$$

That is, a mapping m is allowed by $\mathcal{X}(abs, conc)$ if and only if for each label abs, (1) m is defined over abs if and only if there exists some label $conc$ for which $\mathcal{X}(abs, conc)$ evaluates to true, and (2) m maps abs to such a label $conc$.

Algorithmic considerations. To ensure that the algorithm terminates, the set of expressions that may be constructed using the given grammar is restricted to a finite set, by bounding the domains of data types (e.g., distinct messages and keys in our running example) and the size of expressions. We also assume the existence of a verifier that is capable of checking whether a candidate mapping satisfies a given specification Φ. The verifier implements function $\mathit{verify}(C, P, Q, \Phi)$ which returns *OK* if and only if every mapping allowed by constraint C is valid with respect to P, Q, Φ.

Generalization algorithm. Once we limit the number of candidate expressions to be finite, we can use a brute-force algorithm to enumerate and check those candidates one by one. However, this naive algorithm is likely to suffer from scalability issues. Thus, we present an algorithm that takes a generalization-based approach to identify and prune undesirable parts of the search space. A key insight 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 mapping, 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 Figure 2. The function *synthesize* takes four inputs: processes P and Q, specification Φ, and a user-specified mapping constraint C. It also maintains 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, the algorithm selects some mapping m from C (line 3) and checks whether it belongs to one of the constraints in X (meaning, the mapping is guaranteed to result in an invalid implementation). If so, it is simply discarded (lines 4-5).

Otherwise, the verifier is used to check whether m is valid with respect to Φ (line 7). If so, then *generalize* is invoked to produce a maximal mapping constraint $C_{maximal}$, which represents the largest set that contains $\{m\}$, is contained in C, and is valid with respect to P, Q, Φ (line 9). If, on the other hand, m is invalid (i.e., it fails to preserve Φ), then *generalize* is invoked to compute the largest superset C_{bad} of $\{m\}$ that contains only invalid mappings (i.e., those that satisfy $\neg\Phi$). The set C_{bad} is then added to X and used to prune out subsequent, invalid candidates (line 13).

Constraint generalization. The function $\mathit{generalize}(C', P, Q, \Phi, C)$ computes a maximal set that contains C', is contained within C, and only permits mappings that satisfy Φ. This function is used in two different ways: (1) to identify an undesirable region of the candidate space that should be avoided, and (2) to produce a maximal version of a valid mapping constraint.

The procedure works by incrementally growing C' into a larger set $C_{relaxed}$ and stopping when $C_{relaxed}$ contains at least one mapping that violates Φ. Suppose that constraint C' is represented by a symbolic expression $\mathcal{X}$, which itself is a conjunction of n subexpressions $k_1 \land k_2 \land \ldots \land k_n$, 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 $\mathit{decompose}(C')$ takes the given constraint and returns the set of such subexpressions. The function $\mathit{relax}(C', k_i)$ then computes a new constraint by removing k from C'; this new constraint, $C_{relaxed}$, is a larger set of mappings that subsumes C'.

The verifier is then used to check $C_{relaxed}$ against Φ (line 22). If $C_{relaxed}$ is still valid with respect to Φ, then the implementation decision encoded by k is irrelevant for Φ,

```
1  fun synthesize(P, Q, $\Phi$, C)
2      $X = \{\}$
3      for $m \in C$ do
4          if $\exists\, C_{bad} \in X : m \in C_{bad}$ then
5              skip
6          end
7          $result \leftarrow verify(\{m\}, P, Q, \Phi)$
8          if $result = OK$ then
9              $C_{maximal} \leftarrow generalize(\{m\}, P, Q, \Phi, C)$
10             return $C_{maximal}$
11         else
12             $C_{bad} \leftarrow generalize(\{m\}, P, Q, \neg\Phi, C)$
13             $X \leftarrow X \cup \{C_{bad}\}$
14         end
15     end
16     return none
17 end
18 fun generalize(C', P, Q, $\Phi$, C)
19     $K \leftarrow decompose(C')$
20     for $k \in K$ do
21         $C_{relaxed} \leftarrow relax(C', k)$
22         $result \leftarrow verify(C_{relaxed}, P, Q, \Phi)$
23         if $result = OK \wedge C_{relaxed} \subseteq C$ then
24             $C' \leftarrow C_{relaxed}$
25         end
26     end
27     return $C'$
28 end
```

Fig. 2. An algorithm for synthesizing a maximal mapping constraint.

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). On line 23, we also make sure that $C_{relaxed}$ does not violate the predefined user constraints C.

Example. Let $abs = \mathsf{a.e.}msg$ be a symbolic label that represents Alice sending a message (msg) to Eve, and $conc = msg'.key$ be its corresponding label in the public channel model. Then, one candidate constraint C' for mappings from the high-level to low-level labels can be specified using the following expression:

$$\mathcal{X}(\mathsf{a.e.}msg, msg'.key) \equiv msg = msg' \wedge (msg = \mathsf{s} \Rightarrow key = \mathsf{y}) \wedge (msg = \mathsf{p} \Rightarrow key = \mathsf{x})$$

Suppose that this constraint C' has been verified to be valid with respect to P, Q and Φ. Next, the generalization procedure removes the subexpression $k_1 \equiv (msg = \mathsf{p} \Rightarrow key = \mathsf{x})$ from C', resulting in constraint $C_{relaxed}$ that is represented as:

$$\mathcal{X}(\mathsf{a.e.}msg, msg'.key) \equiv msg = msg' \wedge (msg = \mathsf{s} \Rightarrow key = \mathsf{y})$$

When checked by the verifier (line 22), C' is still considered valid, meaning that the decision encoded by k_1 is irrelevant to the property; thus, k_1 can be safely removed.

However, removing $k_2 \equiv (msg = \mathsf{s} \Rightarrow key = \mathsf{y})$ results in a violation of the property. Thus, k_2 is kept as part of the final maximal constraint expression.

5 Implementation and Case Studies

5.1 Implementation

We have built a prototype implementation[5] of the synthesis algorithm described in Section 4. Our tool uses the Alloy Analyzer [25] as the underlying modeling and verification engine. Alloy's flexible, declarative relational logic is convenient for encoding the semantics of the mapping composition as well as specifying mapping constraints.

[5] The tool, along with the models used in our case studies, is available at `https://github.com/eskang/MappingSynthesisTool`.

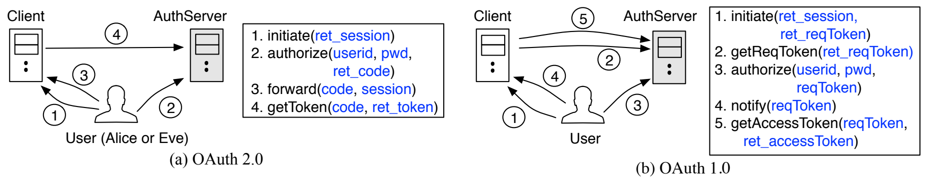

Client
AuthServer
User (Alice or Eve)
1. initiate(ret_session)
2. authorize(userid, pwd, ret_code)
3. forward(code, session)
4. getToken(code, ret_token)
(a) OAuth 2.0
Client
AuthServer
User
1. initiate(ret_session, ret_reqToken)
2. getReqToken(ret_reqToken)
3. authorize(userid, pwd, reqToken)
4. notify(reqToken)
5. getAccessToken(reqToken, ret_accessToken)
(b) OAuth 1.0

Fig. 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 their user ID and password as arguments to AuthServer, which returns ret_code back to User in response.

The analysis engine for Alloy uses an off-the-shelf SAT solver to perform *bounded* verification [25]. In particular, our current prototype is capable of synthesizing mappings to preserve the following types of properties: *reachability* and *safety* properties, which can be expressed in either of the forms $\exists t : t \in T_P \wedge t \in \phi$ (reachability) and $\neg\exists t : t \in T_P \wedge t \notin \phi$ (safety) for some process P and property ϕ.

However, our synthesis approach does not prescribe the use of a particular modeling and verification engine, and can be implemented using other tools as well (such as an SMT solver [11, 12]).

5.2 Case Studies: OAuth Protocols

As two major case studies, we took on the problem of synthesizing valid mappings for *OAuth 1.0 and OAuth 2.0*, two real-world protocols used for *third-party authorization* [24]. The purpose of the OAuth protocol family in general 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 (a *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.

OAuth 2.0 is the newer version of the protocol, while OAuth 1.0 is an older version. 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 [17]. Since both protocols are designed to provide the same security guarantees (i.e., both share common properties), 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.

5.3 Formal Modeling

For our case studies, we constructed the following set of Alloy models: (1) model $P_{1.0}$ representing OAuth 1.0; (2) model $P_{2.0}$ representing OAuth 2.0; (3) model Q representing generic HTTP interactions between a browser and a server, as well as the behavior of a web-based attacker; (4) specification Φ describing desired protocol properties (same for both OAuth 1.0 and 2.0); and (5) mapping constraints $C_{1.0}$ and $C_{2.0}$ representing initial, user-specified partial mappings for OAuth 1.0 and 2.0, respectively. The complete models are approximately 1800 lines of Alloy code in total, and took around 4 man-months to build. These models were then provided as inputs to our tool to solve two instances of Problem 4 from Section 3. In particular, we synthesized a maximal mapping constraint $C'_{1.0}$ such that every $m \in C'_{1.0}$ ensures that $P_{1.0} \|_m Q \models \Phi$. and a maximal mapping constraint $C'_{2.0}$ such that every $m \in C'_{2.0}$ ensures that $P_{2.0} \|_m Q \models \Phi$.

OAuth models ($P_{1.0}, P_{2.0}$). We constructed Alloy models of OAuth 1.0 and 2.0 based on the official protocol specifications [23, 24]. Due to limited space, we give only a brief overview of the models. Each model consists of four processes: Client, AuthServer, and two users, Alice and Eve (the latter with a malicious intent to access Alice's resources).

A typical OAuth 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 their own identity to AuthServer (by providing a user ID and a password) and officially authorize the client to access their resources (authorize). Given the user's authorization, the server then allocates a unique code for the user, and then redirects their back to the client. The user forwards the code to the client (forward), which then can exchange the code for an access token to their resources (getToken).

Like in OAuth 2.0, a typical workflow in OAuth 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 their 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 their resources (getAccessToken).

Specification (Φ). There are two desirable properties of OAuth protocols in general: (1) **Authenticity**: When the client receives an access token, it must correspond to the user who initiated the current protocol session. (2) **Completion**: There exists at least one trace in which the protocol interactions are carried out to completion in the order of steps described in Figure 3. Authenticity is a safety property while completion is a reachability property. The input specification Φ consists of these two properties. Completion is essential for ruling out mappings that over-constrain the resulting implementation and prevent certain steps of the protocol from being performed.

HTTP platform model (Q). Our goal was to explore and synthesize *web-based* implementations of OAuth. For this purpose, we constructed a formal model depicting interactions between a generic HTTP server and web browser. 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 HTTP requests, which share the following signature:

$$\mathsf{req}(method : \mathsf{Method}, url : \mathsf{URL}, headers : \mathsf{List}[\mathsf{Header}], body : \mathsf{Body}, ret\_resp : \mathsf{Resp})$$

The parameters of an HTTP request have their own internal structures, each consisting of its own parameters as follows:

$$\mathsf{url}(host : \mathsf{Host}, path : \mathsf{Path}, queries : \mathsf{List}[\mathsf{Query}]) \quad \mathsf{header}(name : \mathsf{Name}, val : \mathsf{Value})$$
$$\mathsf{resp}(status : \mathsf{Status}, headers : \mathsf{List}[\mathsf{Header}], body : \mathsf{Body})$$

initiate(ret_session) $\longmapsto$ req(GET, http://client.com/initiate?queries, headers, body, ret_resp(OK, [set-cookie: ret_session], body))	forward(code, session) $\longmapsto$ req(POST, http://client.com/forward?queries, headers, body, ret_resp(OK, [ ], body))
authorize(userid, pwd, ret_code) $\longmapsto$ req(POST, http://server.com/authorize?queries, headers, body, ret_resp(Redirect, headers, body))	getToken(code, ret_token) $\longmapsto$ req(GET, http://client.com/getToken?[code], headers, body, ret_resp(OK, [ ], ret_token))

Fig. 4. User-specified partial mappings from OAuth 2.0 to HTTP. 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. 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 their own malicious server and exploit weaknesses in a browser to manipulate the user into sending certain HTTP requests.

Mapping Constraint ($C_{1.0}, C_{2.0}$). 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 for both OAuth protocols; 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). It is reasonable to treat these constraints as given, since they describe decisions that are common across typical web-based OAuth implementations.

Insecure mapping for OAuth 2.0. Let us now give an example of an insecure mapping that satisfies the user-given constraint in Figure 4 but could introduce a security vulnerability into the resulting implementation. Later in Section 5.4, we describe how our tool can be used to synthesize a secure mapping that prevents this vulnerability.

Consider the OAuth 2.0 workflow from Figure 3(a). In order to implement the forward operation, for instance, the developer must determine how the parameters *code* and *session* of the abstract event label 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, $\mathcal{X}_1$:

$$\begin{aligned}\mathcal{X}_1(a,b) \equiv\ & (b.method = \mathsf{POST}) \wedge (b.url.host = \mathsf{client.com}) \wedge \\ & (b.url.path = \mathsf{forward}) \wedge (b.url.queries[0] = a.code) \wedge \\ & (b.headers[0].name = \mathsf{cookie}) \wedge (b.headers[0].value = a.session)\end{aligned}$$

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

This constraint, however, allows a vulnerable implementation where malicious user Eve performs the first two steps of the workflow in Figure 3(a) using her own credentials, and obtains a unique code ($\mathsf{code}_{\mathsf{Eve}}$) from the authorization server. Instead of forwarding this to Client (as she is expected to), Eve keeps the code herself, and crafts their own web page that triggers the visiting browser to send the following HTTP request:

$$\mathsf{req}(\mathsf{POST}, \mathsf{http://client.com/forward?code}_{\mathsf{Eve}}, ...)$$

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 $\mathcal{X}_1$, corresponds to a valid forward event:

$$\mathsf{forward}(\mathsf{code}_{\mathsf{Eve}}, \mathsf{session}_{\mathsf{Alice}}) \mapsto$$
$$\mathsf{req}(\mathsf{POST}, \mathsf{http://client.com/forward?code}_{\mathsf{Eve}}, [(\mathsf{cookie}, \mathsf{session}_{\mathsf{Alice}})], ...)$$

Due to the standard browser logic, the cookie corresponding to $\mathsf{session}_{\mathsf{Alice}}$ is included in every request to client.com. As a result, Client mistakenly accepts $\mathsf{code}_{\mathsf{Eve}}$ as the one for Alice, even though it belongs to Eve, violating the authenticity property of OAuth (this attack is also called *session swapping* [39]).

5.4 Results

Our synthesis tool was able to generate valid mapping constraints for both OAuth protocols. In particular, the constraints describe mitigations against attacks that exploit an interaction between the OAuth logic and security vulnerabilities in a web browser.

OAuth 2.0. The synthesized symbolic mapping constraint for OAuth 2.0 consists of 39 conjuncts in total, each capturing a (conditional) assignment of a concrete HTTP parameter to a constant (e.g., $b.url.path = \mathsf{forward}$) or an abstract OAuth parameter (e.g., $b.url.queries[0] = a.code$). In particular, the constraint captures mitigations against *session swapping* [39] and *covert redirect* [16]. Due to limited space, we omit the full constraint, but instead describe how the vulnerability described at the end of Section 5.3 can be mitigated by our synthesized mapping.

Consider the insecure mapping expression $\mathcal{X}_1$ from Section 5.3. The mapping constraint synthesized by our tool, $\mathcal{X}_2$, fixes the major problem of $\mathcal{X}_1$; namely, 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 mapping expression $\mathcal{X}_2$ synthesized by our tool encodes one form of this solution:

$$\mathcal{X}_2(a,b) \equiv \mathcal{X}_1(a,b) \land (a.session = \mathsf{session}_{\mathsf{Alice}} \Rightarrow b.url.queries[1] = \mathsf{nonce}_0) \land$$
$$(a.session = \mathsf{session}_{\mathsf{Eve}} \Rightarrow b.url.queries[1] = \mathsf{nonce}_1)$$

where nonce_o, $\mathsf{nonce}_1 \in \mathsf{Nonce}$ are constants defined in the HTTP model[6]. In particular, $\mathcal{X}_2$ 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

[6] A nonce is a unique piece of string intended to be used once in communication.

	# total candidates	# explored	# verified	# skipped	Verification		Generali-zation	Total time
					Avg.	Total		
OAuth 1.0	79200	2465	281	2184	2.01	566.05	490.84	1056.89
OAuth 2.0	29400	1453	161	1292	1.88	302.76	1138.85	1441.60

Fig. 5. Experimental results (all times in seconds). "# total candidates" is the total number of possible symbolic mapping expressions; "# explored" is the number of iterations taken by the main synthesis loop (lines 3-15, Figure 2) before a solution was found. Out of these iterations, "# verified" mappings were verified (line 7), while the rest were identified as invalid and skipped (line 5). "Total time" the sum of the Total Verification and Generalization columns) refers to the time spent by the tool to synthesize a maximal constraint.

each session value. $\mathcal{X}_2$ ensures that the resulting implementation satisfies the desired properties of OAuth 2.

OAuth 1.0. The synthesized symbolic mapping constraint for OAuth 1.0 consists of 48 conjuncts in total, capturing how the abstract parameters of the five OAuth 1.0 operations are related to concrete HTTP parameters. The constraint synthesized by our tool for OAuth 1.0 encodes a mitigation against the *session fixation* [15] attack; in short, this mitigation involves strengthening the notify operation with unique nonces (similar to the way the forward operation in OAuth 2.0 was fixed above) to prevent the attacker from violating the authenticity property.

Performance. Figure 5 shows experimental results for the two OAuth protocols[7]. Overall, the synthesizer took approximately 17.6 and 24.0 minutes to synthesize the constraints for 1.0 and 2.0, respectively. In both cases, the tool spent a considerable amount of time on the generalization step to learn the invalid regions of the search space. Note that generalization is effective at identifying and discarding a very large number of invalid candidates; it was able to skip 2184 out of 2465 candidates for OAuth 1.0 ($\approx 88.6\%$) and 1292 out of 1453 for OAuth 2.0 ($\approx 88.9\%$). Our generalization technique was particularly effective for the OAuth protocols, since a significant percentage of the candidate constraints would result in an implementation that violates the completion 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.

6 Related Work

Our approach has been inspired by the success of recent synthesis paradigms such as *sketching* [38, 37, 36], *oracle-guided synthesis* [26] and *syntax-guided synthesis* [3]. Our technique shares many similarities with these approaches in that (1) it allows the user to provide a partial specification of the artifact to be synthesized (in the form of constraints or examples), therefore having the underlying engine *complete* the remaining parts; (2) it relies on an interaction between the verifier, which checks candidate

[7] The experiments were performed on a Mac OS X 2.7 GHz laptop with 8G RAM and MiniSat [13] as the underlying SAT solver employed by the Alloy Analyzer.

solutions, and the synthesizer, which prunes that search space based on previous invalid candidates. Our work also differs in a number of aspects. First, we synthesize mappings from high-level models to low-level execution platforms, which to our knowledge has not been considered before. Second, our approach leverages constraint generalization to not only prune the search space, but also to produce a constraint capturing a (locally) maximal set of valid mappings. Third, our application domain is in security protocols.

A large body of literature exists on *refinement-based* methods to system construction [20, 4]. 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 ϕ when P is implemented using Q.

The task of synthesizing a valid mapping can be seen as a type of the *model merging* problem [8]. This problem has been studied in various contexts, including architectural views [31], behavioral models [32, 6, 40], and database schemas [34]. Among these, our work is most closely related to merging of partial behavioral models [6, 40]. 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 $(P\|_m Q)$ need not be a behavioral refinement of P or Q, as long as it satisfies property ϕ.

Bhargavan and his colleagues presents a compiler that takes a high-level program written using *session types* [22] and automatically generates a low-level implementation [7]. 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. Synthesizing a low-level implementation from a high-level specification has also been studied in the context of data structures [18, 19], although their underlying representation (relational algebra for data schema specification) is very different from ours (process algebra).

A significant contribution of our work is the production of formal models for real-world protocols such as OAuth and HTTP. There have been similar efforts by other researchers in building reusable models of the web for security analysis [1, 5, 14]. As far as we know, however, none of these models has been used for synthesis.

7 Conclusions

In this paper, we have proposed a novel system design methodology centered around the notion of *mappings*. We have presented novel *mapping synthesis problems* and an algorithm for efficiently synthesizing symbolic maximal valid mappings. In addition, we have validated our approach on realistic case studies involving the OAuth protocols.

Future directions include performance improvements (e.g., exploiting the fact that our generalization-based algorithm is easily parallelizable), combining our generalization-based synthesis method with a counter-example guided approach, and application of our synthesis approach to other domains beside security (e.g., platform-based design and embedded systems [35]).

References

1. 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.
2. Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
3. 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 *FMCAD*, pages 1–8, 2013.
4. Ralph-Johan Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6):593–624, 1988.
5. Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF*, pages 247–262, 2012.
6. Shoham Ben-David, Marsha Chechik, and Sebastián Uchitel. Merging partial behaviour models with different vocabularies. In *CONCUR*, pages 91–105, 2013.
7. Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.
8. 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.
9. Suresh Chari, Charanjit S Jutla, and Arnab Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.
10. Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In *CCS*, pages 892–903, 2014.
11. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *lncs*, pages 337–340. Springer, 2008.
12. Bruno Dutertre. Yices 2.2. In *CAV*, pages 737–744, 2014.
13. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.
14. 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.
15. OAuth Working Group. OAuth Security Advisory: 2009.1 “Session Fixation”. `https://oauth.net/advisories/2009-1`, 2009.
16. OAuth Working Group. OAuth Security Advisory: 2014.1 “Covert Redirect”. `https://oauth.net/advisories/2014-1-covert-redirect`, 2014.
17. 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.
18. Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In *PLDI*, pages 38–49, 2011.
19. Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, pages 417–428, 2012.
20. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
21. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
22. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
23. Internet Engineering Task Force. The OAuth 1.0 Protocol. https://tools.ietf.org/html/rfc5849, 2010.

24. Internet Engineering Task Force. OAuth Authorization Framework. http://tools.ietf.org/html/rfc6749, 2014.
25. Daniel Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.
26. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.
27. Eunsuk Kang, Stéphane Lafortune, and Stavros Tripakis. Synthesis of property-preserving mappings. *CoRR*, abs/1705.03618, 2017.
28. Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. Multi-Representational Security Analysis. In *FSE*, 2016.
29. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
30. Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.
31. Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of component and connector models from crosscutting structural views. In *ESEC/FSE*, pages 444–454, 2013.
32. Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve M. Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *ICSE*, pages 54–64, 2007.
33. 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)*, pages 655–659. IEEE, 2011.
34. Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003.
35. Alberto L. Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.
36. Armando Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 4–13. Springer, 2009.
37. Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294. ACM, 2005.
38. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
39. San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *CCS*, pages 378–390, 2012.
40. Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In *SIGSOFT FSE*, pages 43–52, 2004.
41. 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*, pages 365–379, 2012.
42. 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.