ASPIRE: Iterative Specification Synthesis for Security

Kevin Zhijie Chen
UC Berkeley

Warren He
UC Berkeley

Devdatta Akhawe
UC Berkeley

Vijay D'Silva

Prateek Mittal
Princeton University

Dawn Song
UC Berkeley

Abstract

How to perform a systematic security analysis of complex applications is a challenging and open question. Approaches based on formal verification are impeded due to the lack of application specifications. To address this challenge, we propose a framework, called ASPIRE, that enables analysts to automatically synthesize specifications from examples such as application input-output examples and system demonstrations. Our approach starts by synthesizing the initial candidate specifications in a domain specific language that conform to the examples, and iteratively prunes the candidate set by incorporating more user feedback. We implement a prototype of ASPIRE for synthesizing and checking specifications of web applications, although our approach is not limited to web security, and use it in three case studies to demonstrate the discovery of complex vulnerabilities in implementations of real world web applications. Our work is the first to design a general framework that leverages program synthesis techniques for security applications.

1 Introduction

Validating the security of existing applications is a hard challenge. Existing tools approach this problem from two main perspectives. One perspective is that of a top-down approach based on applying model checking and proof systems [2, 5, 6, 22–24] on *manually* constructed models of applications. However, manually writing specifications in a formal language requires significant effort, and remains quite challenging and error-prone. Security analysts wishing to write the specification would need to understand the intricacies of the formal specification language as well as translate complex modern applications into the given formal language. These challenges could lead to the developed model itself being erroneous and inconsistent with the implementation. The second perspective is that of bottom-up approaches that use static analysis techniques to automatically build system models [1, 4, 7] from full system implementations. However, for large applications, the security analyst may only have *partial visibility* into the full implementation. The implementation of a system is typically only partially visible because the code (or binaries) for some components in the system will not be available. For example, a modern web application may rely on services such as Facebook Connect to authenticate users and Paypal for monetary transactions, and a typical mash-up developer lacks access to Facebook or Paypal code. In addition, such approaches are unable to efficiently recognize high-level semantics in the implementation, such as the authentication protocol between Facebook Connect and the third-party application.

Our approach We envision an architecture where a security analyst can specify her intent in terms of application input-output examples or a system demonstration. Our approach, called ASPIRE, provides a middle ground between the two perspectives of manually specifying models, and fully-automated model inference. Instead of manually writing models for each application, we provide a set of common *application-independent* templates (in the form of a *domain specific language* or DSL) whose instantiations become *application-dependent*, and instead of deriving the application models from the implementation, we inductively *synthesize* the models from system execution *traces*.

Figure 1 depicts ASPIRE's architecture. The core of ASPIRE is the encoding of the domain knowledge (including the DSL) for a class of applications, as well as the corresponding synthesis algorithm. The design of the DSL reflects the high-level common patterns the class of applications. The user starts by using examples to demonstrate how the application works. The synthesizer then generate one or more candidate specifications that are consistent with 1) the syntax of the DSL, and 2) the examples. Finally, we set up a feedback loop with the user, to ensure that the system specification meets the user intent and to guarantee correctness and security. A

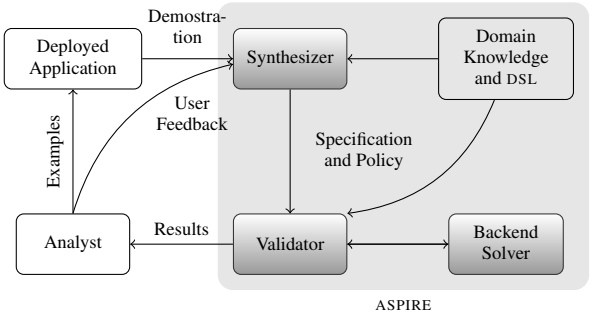

Figure 1: A conceptual workflow of ASPIRE. Users provide example inputs or demonstrations of existing applications. The synthesizer combines these input constraints with domain knowledge of the problem to generate a security specification in a domain specific language. Users can use the generated specification for validation and testing. Alternatively, feedback from the generated specification can produce more inputs.

user feedback can be additional input-output examples, refined demonstrations, or simple hints that point out the errors or confirm the correct statements in the synthesized specification. In this iterative fashion, the user can generate and refine system models or policies.

We envision a number of uses for such an automatically synthesized specification. First, system builders and users can use classic analysis and verification techniques on the automatically generated formal specifications; thereby achieving much higher assurance in the system. Second, a compiler can automatically translate the specification into a (partial) implementation. Finally, the results loop, which generates examples and natural language descriptions of formal specifications, can help better understand existing systems. This is in addition to generating and verifying system specifications using examples. We propose to design and build a generic framework that is able to leverage synthesis technologies in a broad range of security applications, such as web security, cloud security, mobile security, and even network intrusion detection. While our goals are broad, for concrete exposition of our ideas, we explain our vision using a domain-specific concretization for web security and a few use cases.

Our work makes the following contributions.

1. A general approach for synthesizing security specifications from system execution traces and domain specific languages.
2. A DSL to represent certain semantics of web applications, and a new algorithm for the synthesis of web protocol models (in terms of the DSL) from system executions and analyst feedback.
3. Three proof-of-concept case studies to demonstrate how our system can efficiently detect real-world, *session integrity* vulnerabilities, including previously unknown vulnerabilities.

```
GET /login HTTP/1.1
Host: bodgeitstore.com

HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=7ffa4512

<form method="post" action="/login">
<input type="hidden" name="csrftoken" value="3eff8527">
<input type="text" name="username">
<input type="password" name="password">
<input type="submit" name="submit" value="login">
</form>

POST /login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: session=7ffa4512
Host: bodgeitstore.com

csrftoken=3eff8527&username=user1&password=secretpwd&submit=login

HTTP/1.1 200 OK
Content-Type: text/html

<b>Welcome!</b>
```

Listing 1: Example traces from the Bodgeit Store

2 ASPIRE for the Web

To demonstrate the concrete benefits of our approach, we present ASPIRE, a prototype system that aims to find vulnerabilities in web applications.

Running Example We use the Bodgeit Store [19] as our running example. It is an online shopping web application that allows users to register an account, login to an account, manage shopping carts, and make purchases. It is deliberately made vulnerable and used for teaching web security. Listing 1 is an example of some messages from its protocol.

The design of ASPIRE consists of three main components. First, we define a DSL for web protocols to control and customize the search space in vulnerability discovery. Second, we introduce a new algorithm for synthesizing specification describing a web protocol from execution traces (as the system demonstration) and analyst feedback. Third, we integrate a formal verification tool (as the validator) with the inferred model to construct an end-to-end system for vulnerability detection. ASPIRE accommodates interactive system analysis: the specification can be visualized as a *message sequence chart* for human inspection, and the analyst can interact with the system via additional example execution traces, and provide *hints* as a feedback to the vulnerabilities and the issues ASPIRE identifies.

To solve the problem of synthesizing specifications for vulnerability discovery, we must solve two problems. The *representation problem* is to design a DSL that captures the subtleties of web protocols while still remaining high-level for human understanding. The *algorithmic problems* are to construct a protocol model, find vulnerabilities, and update the model using analyst feedback.

The details of the algorithm depend on the representation, and hence preclude an off-the-shelf solution.

2.1 Designing a Representation

We design a language, called the Model Description Language (MDL), with primitives chosen to enable succinct descriptions of web protocols. These primitives have a precise semantics and compile into ALLOY, the input language for the ALLOY model checker.

The design of the MDL is motivated by the web model formally defined in previous work [2, 5, 6]. The modeling of a web application's logic consists of two components, the application-independent component and the application-specific component. The application-independent component, i.e., the semantics of the MDL, defines a generic web model, including the definition of HTTP servers and clients, the same origin policy, the use of cookies, etc. The application-specific component, i.e., a specification written in MDL, uses the basic definitions in the application-independent component to describe the logic of a web protocol. For example, Figure 2 lists the specification derived from the Bodgeit Store's example traces in Listing 1. In ASPIRE, MDL models the following web concepts:

Basic concepts A *web application* is a distributed system based on the HTTP protocol. We refer to the participants of a web application as *endpoints*. A *client endpoint* (*client* for short) is typically a browser and a *server endpoint* (*server* for short) is a web principal represented by its web origin. A *message* is either an HTTP request or an HTTP response. Messages are abstractly represented as mappings from *keys* to *values*. A *web protocol* is a specification of the sequences of messages exchanged between endpoints and the invariants imposed on these messages. Each of these concepts will be translated into an ALLOY signature in our base model.

Interaction The set of endpoints consists of a benign client, a malicious client, a malicious server and a set of benign servers defined by the synthesized MDL specification. The *benign client* can send arbitrary request permitted by the rule of the server. When it receives a redirection response, it immediately sends a request with the URL specified by the redirection. For each *benign server*, it only accepts requests and sends responses according the synthesized specification.

Threat model We consider the web attacker model and session integrity formally defined in Akhawe et al. [2]: A *web attacker* is a malicious principal who controls a web server visited by the user. Intuitively, the web attacker can be thought as having "root access" to this web server, and is able to retrieve arbitrary information contained in the request and send arbitrary response to the user. In addition, the web attacker can send arbitrary HTTP requests to the benign servers.

Security Policy Typical attacks on web protocols violate the *session integrity*. A session integrity condition states that the attacker should not be able to cause benign servers to undertake potentially sensitive actions. CSRF attacks are typical violations of the session integrity policy. For example, a login CSRF attack violates the login session integrity by directly logging in the user without the initial login page. In our running example, the invariant at line 18-20 prevents login CSRF attacks.

These definitions are reflected in the ALLOY base model. For example, listing 2 shows the ALLOY predicate that checks if an HTTP request belongs to a CSRF attack. And the model checker will check if such attack could happen in our synthesized specification.

```
1 pred isCSRF[r: HTTPRequest] {
2    (some r.prev and r.prev in MaliciousRedirectionResponse)
3    (r.from = VictimClient)
4    (r.to in VictimServer))
5    some (r.payload − r.cookies)
6    attackerCanLearn(r.payload − r.cookies)
7 }
```

Listing 2: The ALLOY code that checks for CSRF attacks

Since MDL specifications are translated into ALLOY, embedded in the base model, and verified, the base model also determines the semantics of MDL.

```
 1 servers: bodgeit;
 2 init:
 3     bodgeit knows t1,t2;
 4     client knows t3,t4;
 5 messages:
 6     request(server=bodgeit, type=req-helo),
 7     response(server=bodgeit, type=resp-helo,
 8         fields=(jsid in setcookie, csrf in body)),
 9     request(server=bodgeit, type=req-login,
10         fields=(rcsrf in urlparam, rjsid in cookie,
11                 username in urlparam, password in urlparam)),
12     response(server=bodgeit, type=resp-login);
13 invariants:
14     resp-helo.jsid isa t1;
15     resp-helo.csrf isa t2;
16     req-login.username isa t3;
17     req-login.password isa t4;
18     forall m1:resp-helo, m2:req-helo {
19         m1.jsid == m2.rjsid <=> m1.csrf == m2.rcsrf;
20     }
```

Figure 2: The illustrative MDL specification for a website's login process.

2.2 Algorithmic Components

Next, we present how the specification is synthesized, verified, and refined.

Specification synthesis The first algorithmic problem is to construct a protocol model from HTTP traces. Since our model is expressed in MDL, model construction can be viewed as a *specification synthesis* task. Specifically, the set of all MDL specifications defines a search space, and synthesis from examples seeks to find specifications that generate and generalize those sample executions.

The specification synthesis task is split into a few subtasks that infers different aspects of the specification, including the set of endpoints, the initial data known to the each endpoint, the formats of the messages sent and received by the endpoints, and the invariants over the messages and their payloads.

Vulnerability discovery The second algorithmic problem is to discover vulnerabilities in the protocol model. Our system includes generic description of CSRF vulnerabilities in web protocols. We compile a MDL specification together with a vulnerability description into an ALLOY model, and reduce the vulnerability discovery problem to a model checking problem. The ALLOY model reflects the semantics of the MDL defined in Section 2.1. The verification condition enforces that none of the traces permitted by the specification forms a CSRF attack.

The result of the verification can be `Safe`, `Timeout` or `Vulnerable`. For all the three cases, a synthesized protocol will be visualized to the analyst in a message sequence chart. Additionally, in the case when the protocol is vulnerable, a trace will be presented to the analyst as the demonstration of the attack.

Feedback and refinement The third algorithmic problem is to incorporate analyst feedback to update either the protocol model or the vulnerability description used by the system. During this phase, the analyst inspects the results by reading the message sequence charts or replaying the attack trace to the actual web service, and reaches a conclusion on the correctness of the synthesized protocol and whether the attack trace is spurious. If the synthesis is not accurate or the attack trace is spurious, the analyst provides more hints to the synthesizer.

Our system accepts three types of hints: *Input hints*, in which a new input value is provided as an alternative to the values that are previously constant, or an annotation is provided to ignore a message field; *Scope hints*, which are URLs that should be considered or ignored during the synthesis; and *Target hints*, which are messages that should be considered or ignored in the verification.

3 Case Studies

We use ASPIRE to identify vulnerabilities in three real world applications. We rediscover two (previously found manually) vulnerabilities and discover four previously unknown vulnerabilities. We summarize the configuration and the performance of each iteration in Table 1. We performed all the experiments on a desktop machine with an Intel i5 670 3.4GHz CPU and 8GB memory. The performance is moderate considering it is an *offline* analysis that does not interrupt the web services.

The CAS Protocol First, we analyze the Central Authentication Service protocol (CAS) [17]. The CAS protocol was originally developed at Yale University, and at least eighty universities currently deploy it [14]. Akhawe et al. manually wrote down the protocol model and identified a session fixation attack (later fixed) [2].

Name	#Servers	New Hints	#Msgs	Verif. Time (s)	Vuln.?
CAS	2	None	12	7.17	Y(New)
		Target(-)	12	41.71	Y
		None	12	>7200	N
NMP	1	None	8	7.20	Y(New)
		Target(-)	8	9.53	N
		Input(+)	8	8.16	Y
GOV	2	None	48	>7200	N
		Scope(-)	24	699.91	Y(New)
		Target(-)	24	2399.77	Y(New)

Table 1: Configuration and performance of the case studies. The first column lists the names of our case studies. The second column lists the number of benign servers involved. The third column lists the new hints provided by the analyst in each iteration. For each type of the hint, we use "+" to indicate a positive answer, and "-" to indicate a negative answer. The fourth column lists the number of message types in the synthesized model. The fifth column lists the verification time. We bound the verification to take up to 7200 seconds. The last column lists whether we find any vulnerabilities. The protocol synthesizer terminates within 5 seconds in all the case studies.

To validate the fidelity of ASPIRE, we attempted to recreate and automatically identify this vulnerability. We captured example traces at our university by logging into a class registration system (twice, with different user accounts) using the CAS protocol. We manually removed the fix to the aforementioned vulnerability. Figure 3 shows the synthesized vulnerable protocol. The protocol is equivalent to the one manually written [2].

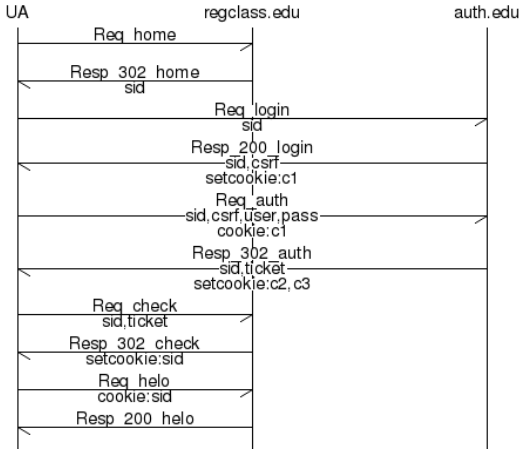

Figure 3: The vulnerable CAS protocol

In the initial iteration, we embed the vulnerable protocol into our base model and check for vulnerabilities. The initial attack trace returned by the model checker is a valid session-riding attack. Interestingly, this attack was missed by prior manual analysis.

In order to search for other vulnerabilities, we provide a negative target hint to exclude this vulnerability, and

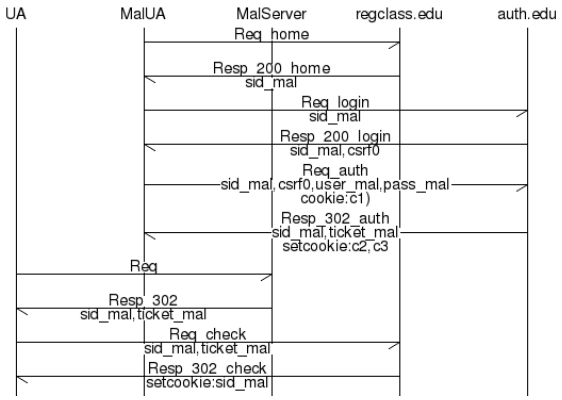

Figure 4: The second attack trace for the CAS protocol

continue the search for more attacks. The verification with the modified vulnerability condition returns an attack trace as shown in Figure 4. The attack trace says that the attack can authenticate with the authentication server first and get a ticket. Instead of redirecting to *Req_check* in the attacker's browser, the attacker sends the link to a benign user who ends up logging in as the attacker on the user's browser. If the user is not aware of this, he or she could end up registering for classes or paying tuition for the attacker. This finding validates the performance of ASPIRE. Recall that the vulnerability was previously found using significant manual analysis. Furthermore, the manual analysis missed the vulnerability discovered in the initial iteration.

NeedMyPassword.com NeedMyPassword.com is an online password manager. We found 1 new login CSRF vulnerability and 1 previously known CSRF vulnerability. We note that previous work *manually* identified the second CSRF vulnerability that we found [16]. This demonstrates the power of our approach; not only was ASPIRE able to rediscover known vulnerabilities from system execution traces, but its systematic analysis was also able to identify new vulnerabilities in protocols that have been manually vetted.

Govtrak.us and Facebook Connect Govtrak.us is a website for easily tracking the activities of the United States Congress. It provides some social features where the user can associate his or her govtrak.us account with a Facebook account, and also login with Facebook accounts. ASPIRE successfully synthesized two attack traces that exploit two different weaknesses during the process of binding a Facebook account to a govtrak.us account. At a high-level, these vulnerabilities are similar to the ones found in the CAS protocol, since they are both violations of the session integrity policy, and both protocols are single-sign-on protocols. However, we want to emphasize that our base model and the DSL do not have any special treatment for single-sign-on or authorization protocols in general. The synthesis algorithm manages to recognize the patterns in the traces and generalize them into protocol specifications for the CAS protocol and the Facebook Connect protocol.

4 Related Work

At the core of our system is the synthesis of the protocol description in a DSL. Program synthesis aims to develop technologies that can translate expressions of user intent into programs. For example, researchers have recently proposed techniques to automatically synthesize programs using input-output examples [11, 13], system predicates over input-output [12, 21], program template structure and constraints [20], natural language [8], and user demonstration [15]. Gulwani et al. provided a comprehensive overview of such techniques [10].

Our approach is motivated by existing work on inductive model generation. We aim to generalize from existing work and build a modular and extensible security specification synthesis framework. Wang et al. performed a series of mostly manual security analysis on cashier-as-a-service based web stores [23], single-sign-on web services [22] and protocol SDKs [24]. The latter also demonstrates a systematic process of the interaction between the security analyst and a formal analysis tool. AuthScan [3] is a system for automatic extraction and checking of web authentication protocols. It performs differential analysis on traces generated by symbolic execution and testing to infer the invariants in the protocols. Pellegrino and Balzarotti [18] propose a black-box technique that identifies a number of behavioral patterns from network traces and generates test cases. Invariant detectors like Daikon [9] also inductively generate invariants from traces.

5 Conclusion

In this paper, we present ASPIRE, an interactive framework that enables the modeling and verification of program implementations without access to the source code. We illustrate how our system works by presenting a prototype implementation for the security analysis of web applications. ASPIRE uses execution traces and analyst feedback to construct and refine protocol models. Our key insight is to leverage modern program synthesis techniques in inferring the application models. Our system eases the burden of manually translating the implementations of applications into formally verifiable models. Using several proof-of-concept case studies, we have demonstrated how ASPIRE can discover both previously known vulnerabilities as well as new vulnerabilities from real world applications. Our research is the first step to explore the benefits of program synthesis and domain specific languages for enhancing application security.

Acknowledgement: This work is supported by the NSF under Grant No. 1409915 and 1409415.

References

[1] AIZATULIN, M., GORDON, A., AND JÜRJENS, J. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 331–340.

[2] AKHAWE, D., BARTH, A., LAM, P. E., MITCHELL, J., AND SONG, D. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE* (2010), IEEE, pp. 290–304.

[3] BAI, G., LEI, J., MENG, G., VENKATRAMAN, S., SAXENA, P., SUN, J., LIU, Y., AND DONG, J. Authscan: Automatic extraction of web authentication protocols from implementations. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2013).

[4] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 203–213.

[5] BANSAL, C., BHARGAVAN, K., DELIGNAT-LAVAUD, A., AND MAFFEIS, S. Keys to the cloud: formal analysis and concrete attacks on encrypted web storage. In *Principles of Security and Trust*. Springer, 2013, pp. 126–146.

[6] BANSAL, C., BHARGAVAN, K., AND MAFFEIS, S. Discovering concrete attacks on website authorization by formal analysis. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th* (2012), IEEE, pp. 247–262.

[7] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement. In *Computer aided verification* (2000), Springer, pp. 154–169.

[8] COZZIE, A., AND KING, S. T. Macho: Writing programs with natural language and examples. Tech. Rep. 2142.33791, University of Illinois at Urbana-Champaign, 2012.

[9] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1 (2007), 35–45.

[10] GULWANI, S. Dimensions in program synthesis. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Austin, TX, 2010), FMCAD '10, FMCAD Inc, pp. 1–2.

[11] GULWANI, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 317–330.

[12] GULWANI, S., KORTHIKANTI, V. A., AND TIWARI, A. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 50–61.

[13] HARRIS, W. R., AND GULWANI, S. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 317–328.

[14] JASIG. The CAS protocol deployment. Online, 2010. http://www.jasig.org/cas/deployments.

[15] LAU, T., WOLFMAN, S. A., DOMINGOS, P., AND WELD, D. S. Programming by demonstration using version space algebra. *Machine Learning 53*, 1-2 (2003), 111–156.

[16] LI, Z., HE, W., AKHAWE, D., AND SONG, D. The emperors new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014).

[17] MAZUREK, D. The CAS protocol. Online, 2005. http://www.jasig.org/cas/protocol.

[18] PELLEGRINO, G., AND BALZAROTTI, D. Toward black-box detection of logic flaws in web applications. In *Network and Distributed System Security (NDSS) Symposium* (February 2014), NDSS 14.

[19] PSIINON. The bodgeit store. Online, 2010. `https://code.google.com/p/bodgeit/`.

[20] SRIVASTAVA, S., GULWANI, S., CHAUDHURI, S., AND FOSTER, J. S. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 492–503.

[21] SRIVASTAVA, S., GULWANI, S., AND FOSTER, J. S. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2010), POPL '10, ACM, pp. 313–326.

[22] WANG, R., CHEN, S., AND WANG, X. Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 365–379.

[23] WANG, R., CHEN, S., WANG, X., AND QADEER, S. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 465–480.

[24] WANG, R., ZHOU, Y., CHEN, S., QADEER, S., EVANS, D., AND GUREVICH, Y. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of the USENIX Security Symposium* (2013), USENIX.