AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations*

Guangdong Bai*, Jike Lei*, Guozhu Meng*, Sai Sathyanarayan Venkatraman*,
Prateek Saxena*, Jun Sun†, Yang Liu‡, and Jin Song Dong*
*National University of Singapore
†Singapore University of Technology and Design
‡Nanyang Technological University

Abstract

Ideally, security protocol implementations should be formally verified before they are deployed. However, this is not true in practice. Numerous high-profile vulnerabilities have been found in web authentication protocol implementations, especially in single-sign on (SSO) protocols implementations recently. Much of the prior work on authentication protocol verification has focused on theoretical foundations and building scalable verification tools for checking manually-crafted specifications [17, 18, 44].

In this paper, we address a complementary problem of automatically extracting specifications from implementations. We propose AUTHSCAN*, an end-to-end platform to automatically recover authentication protocol specifications from their implementations.* AUTHSCAN *finds a total of 7 security vulnerabilities using off-the-shelf verification tools in specifications it recovers, which include SSO protocol implementations and custom web authentication logic of web sites with millions of users.*

1 Introduction

Web authentication mechanisms evolve fast. Many web sites implement their own authentication protocols and rely on third-party mechanisms to manage their authentication logic. For example, recent single-sign on (SSO) mechanisms (e.g., Facebook Connect, SAML-based SSO, OpenID and BrowserID) have formed the basis of managing user identities in commercial web sites and mobile applications. For example, OpenID currently manages over one billion user accounts and has been adopted by over 50,000 web sites, including many well-known ones such as Google, Facebook and Microsoft [5]. As another example, Facebook Connect is employed by 2 million web sites and more than 250 million people reportedly use it every month as of 2011 [7]. Ideally, authentication protocols should be formally verified prior to their implementations. However, majority of web sites do not follow this principle. Authentication protocols have historically been hard to design correctly and implementations have been found susceptible to logical flaws [31, 41]. Web authentication protocols are no exception—several of these implementations have been found insecure in post-deployment analysis [16, 29, 39, 42].

There are three key challenges in ensuring that applications authenticate and federate user identities securely. First, most prior protocol verification work has focused on checking the high-level protocol specifications, not their implementations [13,21,44]. In practice, however, checking implementations is difficult due to lack of complete information, such as missing source code of some protocol participants. Second, verifying authentication using off-the-shelf tools requires expert knowledge and, in most prior work, conversion of authentication protocol specifications to verification tools has been done manually. However, several custom authentication protocols are undocumented. As new protocols emerge and the implementations of existing protocols evolve, manual translation of every new protocol becomes infeasible. Moreover, manual translation is tedious and can be error-prone. Finally, the authentication of the communication between protocol participants often goes beyond the initial establishment of authentication tokens, which the high-level specifications dictate. In practice, checking the end-to-end authentication of communication involves checking if the authentication tokens are actually used in all subsequent communications and making sure they are not sent on public communication channels or stored in persistent devices from which they can leak. Techniques to address these practical problems of existing implementations are an important area which has received relatively lesser attention.

Our Approach. In this paper, we present a frame-

*Prateek Saxena, Jun Sun and Yang Liu have contributed equally to this work.

work called AUTHSCAN to automatically extract the formal specifications of authentication protocols from their implementations. Then, these specifications are directly checked for authentication and secrecy properties using off-the-shelf verification tools [10, 18, 22]. AUTHSCAN can automatically confirm the candidate attacks generated by the verification tools and report the true positives (confirmed attacks) in most cases we study. In some cases, AUTHSCAN does not know the attacker's knowledge set enough to generate confirmed attacks — in such cases, it generates security warnings containing precise communication tokens that need to be manually reviewed by the security analyst.

We design an intermediate language TML to bridge the gap between the detailed implementation of an authentication protocol and its high level semantics that can be used by the verification tools. We show that TML is sufficient to capture the communications between protocol participants and their internal actions. AUTHSCAN assumes no knowledge of the protocol being inferred and does *not* require the full source code of the implementation. We propose a refinement method to deal with partial availability of the code implementing the protocol (e.g., if the code located on a web server is not available). It starts with an initial abstraction of the protocol specification, and iteratively refines the abstraction until it reaches a fixpoint. To perform this refinement, we propose a novel *hybrid inference* approach to combine a whitebox program analysis with a blackbox *differential fuzzing* analysis. In particular, the whitebox analysis performs dynamic symbolic analysis on the available code to extract precise data semantics and the internal actions of the protocol participants. The blackbox analysis infers the protocol implementation by probing the protocol participants and comparing the changes in their response. Our final inferred specification in TML can be directly translated into modeling languages used by off-the-shelf verification tools and can be configured to verify against a variety of attacker models [17, 24].

Our techniques focus on recovering as much protocol semantics as possible from dynamic executions of the protocol; we do not aim to find complete specifications. Instead, we aim to recover fragments of the protocol with enough precision to find interesting logic flaws. We apply AUTHSCAN to study several real-world web sites, including three popular SSO protocols — Facebook Connect Protocol (2 web sites), Browser ID (3 web sites) and Windows Live Messenger Connect (1 web site). We also test several standalone web sites which implement their custom authentication logic and have millions of users sharing personal information. AUTHSCAN successfully recovers precise (but partial) models of their authentication logic, and formally verifies their authentication and secrecy properties against a broad range of attacker models. We have found 7 security flaws in these implementations without their prior knowledge—one of these was found independently by a concurrent work [33] and the remaining are previously unknown. In particular, we find two flaws in Facebook Connect Protocol and one flaw in BrowserID, which arise because the freshness of messages is not guaranteed in the protocol implementations. An attacker is thus able to perpetrate replay attacks to acquire unauthorized authentication credentials. Several other vulnerabilities are due to unsafe implementation errors in creating and maintaining secrecy of authentication tokens. For example, we find that a web site employing Windows Live Messenger Connect grants the end user a publicly known value as a credential after the user has been authenticated to Windows Live.

Contributions. We make the following main contributions in this paper:

- **Automatic Extraction Techniques.** We propose automatic techniques to extract the authentication protocol specifications from their implementations. Our approach works with only minimal number of user inputs (Section 2.3) and reasonable assumptions (Section 3.2), without requiring *any* knowledge of the protocol. Our techniques gracefully adjust the precision of the inferred protocol based on how much source code implementing the protocol is visible to the analysis.
- **End-to-end System**. We build AUTHSCAN, an end-to-end system that embodies these techniques. AUTHSCAN is designed to be extensible and configurable—it can utilize several off-the-shelf verification tools (ProVerif [18] or PAT [38]), and can be extended to model different attack models.
- **Practical Results.** We apply our approach to several real-world web sites, including several using important SSO protocols like Facebook Connect Protocol, BrowserID and Windows Live Messenger Connect. We successfully find 7 security flaws in their implementations.

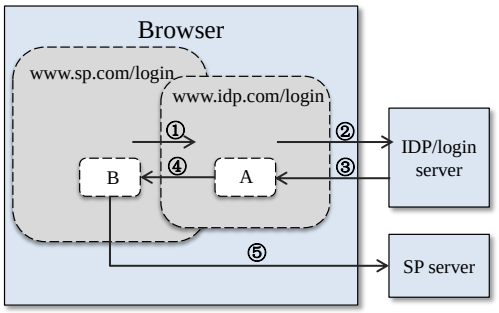
2 Challenges & Overview

Security analysts often need to guarantee the correctness of authentication protocol implementations without having complete access to the source code. In this section, we explain the problem and its challenges with an example.

2.1 A Running Example

Consider one execution of a hypothetical single-sign on (SSO) protocol (similar to Facebook Connect) as shown in Figure 1-(a). In our example, Alice wants to authenticate herself to a service provider (SP) web site hosted at `sp.com` by using her login credentials with an identity provider (IDP) hosted at `idp.com`[1]. This example shows

[1]One sample IDP is `facebook.com` in Facebook Connect and one sample SP is `cnn.com` which uses the Facebook Connect protocol.



(a) The process of Alice authenticates herself to the SP though the IDP

①	*SP_C* → *IDP_C*	*spid, spDomain, next*
②	*IDP_C* → *IDP_S*	*spid, spDomain, sessionID, CSRFToken*
③	*IDP_S* → *IDP_C*	*uEmail, authToken,* $\{uEmail, authToken\}_{k_{IDP}^{-1}}$
④	*IDP_C* → *SP_C* *(next)*	*uEmail, authToken,* $\{uEmail, authToken\}_{k_{IDP}^{-1}}$
⑤	*SP_C* → *SP_S*	*uEmail, authToken*

(b) Communication actions of the participants (*IDP_C*: IDP client code, *IDP_S*: IDP server, *SP_C*: SP client code, *SP_S*: SP server)

```
②  1  GET  https://www.idp.com/login?spid=SID&spDomain=sp.
   2       com&redirect_url=http://www.idp.com/granter?next=
   3       http:// www.sp.com/login
   4  Host: www.idp.com
   5  Referer: https://www.idp.com/login
   6  Cookie: sessionID=0x12345678
   7  ----------------------------------------
   8          CSRFtoken=sLd2f93
③  9  HTTP/1.1 200
  10  Set-Cookie: cookie1=87654321; domain=.idp.com
  11    ----------------------------------------
  12  <body onload=foo()> <script>                          ┐
  13    var domain="http://www.sp.com/login";               │
  14    var authToken="3fa09d24a3ce";                       │
  15    var uEmail="alice@idp.com";                         │
  16    var idpSign="2oOs5u29erIas…";                       │ A
  17    function foo(){                                     │
  18      var message=uEmail+"&"+authToken+"&"+idpSign ;    │
  19      window.postMessage(domain, message);  }           │
  20  </script> </body>                                     ┘
B 21  window.addEventListener('message',function(event) {
  22    var uEmail=extractUser(event.data);
  23    var authToken=extractToken(event.data);
  24    var idpSign=extractSign(event.data);
  25    var data=uEmail+"&"+authToken;
  26    var idpPubKey=loadPubKey();
  27    if(verify(data, idpSign, idpPubKey)){
  28      var message=uEmail+"&" +authToken;
  29      var request = $.ajax({url: login, data: { token: message}});}
  30    else
  31    {…}},false);
```

(c) Parts of exchanged HTTP message and client code

Figure 1: An SSO example: Alice authenticates herself to the SP (*sp.com*) by using her login credentials with the IDP (*idp.com*). The circled numbers indicate the login process, and the capital letters stand for client code.

that much of the communication between the IDP and the SP occurs through the web browser (using `postMessage` between client-side `iframes`), which is similar to real-world protocols [27, 42]. This enables security analysts to analyze protocol behaviors.

The authentication protocol, which the security analyst aims to infer, is as follows:

- **Step** ①: When Alice visits the SP's site and initiates the intent to authenticate, the client-side SP code sends the pre-registered ID and domain of the SP to the IDP's `iframe`. The fact that each SP is pre-registered with the IDP is not known to the security analyst by observing this protocol execution.
- **Step** ②: Assuming that Alice has already logged into the IDP, the IDP generates an HTTP request to its backend server. The request contains a nonce (anti-CSRF) and the session ID of Alice's ongoing web session with the IDP.
- **Step** ③: The IDP replies with Alice's registered email identity `uEmail` and an authentication token `authToken`, which authorizes all access to Alice's personal information stored at the IDP. The IDP creates a cryptographic signature over the terms `uEmail` and `authToken` as an authentication credential to be verified by the SP.
- **Step** ④: Client-side IDP code (code A in Figure 1-(c)) relays the HTTP data received in step ③ to the SP's `iframe`.
- **Step** ⑤: Client-side SP code (code B in Figure 1-(c)) verifies that the signature is valid and extracts the `uEmail` and `authToken`. The SP's `iframe` sends Alice's identity and `authToken` back to the SP's server. This allows the SP's server to access Alice's information stored at the IDP, and allows the IDP to log all SP's actions on Alice's data for audit (not shown).

The security analyst can only observe the network traffic and code execution at the browser end; the server-side logic of the protocol participants is not available for analysis.

Security Flaws. The protocol has several vulnerabilities. We only describe three of them and they can be found automatically if the protocol can be inferred precisely:

- **Man-in-the-middle (MITM) Attack**. The protocol is susceptible to several MITM attacks by a web attacker. For example, consider the target of the `postMessage` call in the client-side code (line 19). This target is derived from an HTTP parameter called `next` (at line 2 of Figure 1-(c)). A malicious SP, say Eve, can change the `next` parameter to its own domain, leaving the `spid` parameter as it is. In this attack, the token granted to the `sp.com` is actually sent to Eve by code labeled as **A** in step ④. This attack is

similar to a recently reported real-world attack on the site `zoho.com` employing Facebook Connect [42].

- **Replay Attack**. The protocol is susceptible to a replay attack, as the IDP's server does not use any nonce or timestamp to guarantee the freshness of the authentication token `authToken`. If a malicious SP obtains the signed assertion in step ④, it can replay the message to `sp.com` in a new web session and log in as Alice.
- **Guessable Tokens**. Even if the authentication token is kept secret by carefully using only secure (private) communication channels, additional problems can exist. For example, `authToken` remains constant across all of Alice's sessions, which is not apparent from observing a single protocol run. We refer to such tokens as *long-lived* tokens. Long-lived tokens may be used in replay attacks. Similarly, if the IDP uses a weak or guessable scheme to generate authentication tokens, such as a sequentially incrementing counter, an attacker can precisely guess the tokens used in other web sessions.

2.2 Challenges

This example shows that implementation-dependent security properties need to be checked in real web applications, where the formal specifications are required. In the following, we list a number of practical challenges in inferring specifications from their implementations.

Inferring Semantics. A key challenge is to infer the precise semantics of data elements exchanged in the communication. For example, it is important to know that `authToken` remains constant across all of Alice's sessions with the IDP and does not include a nonce or a timestamp. Inferring this information is critical to discover the replay attack in the protocol. Similarly, identifying that the communication target in ④ is not a fixed domain but instead a variable derived from the HTTP parameter `next` is crucial to find the MITM attack. These semantics are not obvious from the values observed in one message or even in one execution of the protocol.

Partial Code. Only the part of the protocol implementation that executes in the web browser is visible for analysis. For instance, we can infer using whitebox analysis over the client-side code that `idpSign` is a cryptographic signature of `uEmail` and `authToken` under the IDP's private key. This allows us skip generating random guesses about whether it is possible to forge the (`uEmail`, `authToken`) pair by the attacker. This can significantly improve the precision, which we discussed in Section 6. In other cases, the exact relationship between data elements is not directly available via whitebox analysis. For example, no client-side code reveals whether `authToken` is tied to `sp.com` or is the same for all SPs registered with the IDP. Our analysis needs to infer if there is a one-to-one relation between them.

Redundant Message Elements. Numerous HTTP data elements are contained in the HTTP traces, but most of them are irrelevant to the authentication protocol. The cookie *cookie1* (line 10 in Figure 1-(c)) is one of such examples. Including redundant element when using off-the-shelf verification tools can significantly increase the verification time or even lead to a non-termination. One of the challenges for scalability is to identify and eliminate irrelevant parameters systematically from the traces.

2.3 AUTHSCAN Overview

To overcome these challenges, we develop a tool called AUTHSCAN which requires no prior knowledge of the protocol. AUTHSCAN is a system that aids security analysts. It takes the following three inputs.

- **Test Harness**. The security analyst provides AUTHSCAN with at least one implementation of the protocol and provides login credentials (such as username and password) of at least two test accounts. The analyst can optionally provide additional test cases involving many different users and/or different participants (such as different SPs) to utilize AUTHSCAN's full capability—the more test cases, the more precise is the inferred protocol.
- **Protocol Principals & Public keys**. In each test case, the analyst specifies the principals relevant to the protocol, such as the SP, the IDP and the user being authenticated in the running example. In addition, AUTHSCAN takes as inputs the interface APIs (web URIs) that can be queried to obtain public keys of principals involved in the protocol. For instance, JavaScript function `loadPubKey` at line 26 in the running example internally makes an `XmlHttpRequest` (not shown) to retrieve the public key of the IDP; such web interfaces need to be identified by the analyst.
- **Oracle**. AUTHSCAN generates new protocol executions internally during testing. For each internal run generated, AUTHSCAN needs to query a test oracle that indicates whether authentication is successful or not. For AUTHSCAN, this is specified as an HTTP request that AUTHSCAN can make to verify a successful completion. In the running example, AUTHSCAN can generate an HTTP request to access Alice's personal information at the IDP using `authToken` to check if the protocol run succeeds.

Output. AUTHSCAN produces two outputs. First, it produces a specification of the inferred protocol, which can act as a starting point for a variety of manual and automatic analysis [17]. Second, it produces a vulnerability report for all the attacks that it finds.

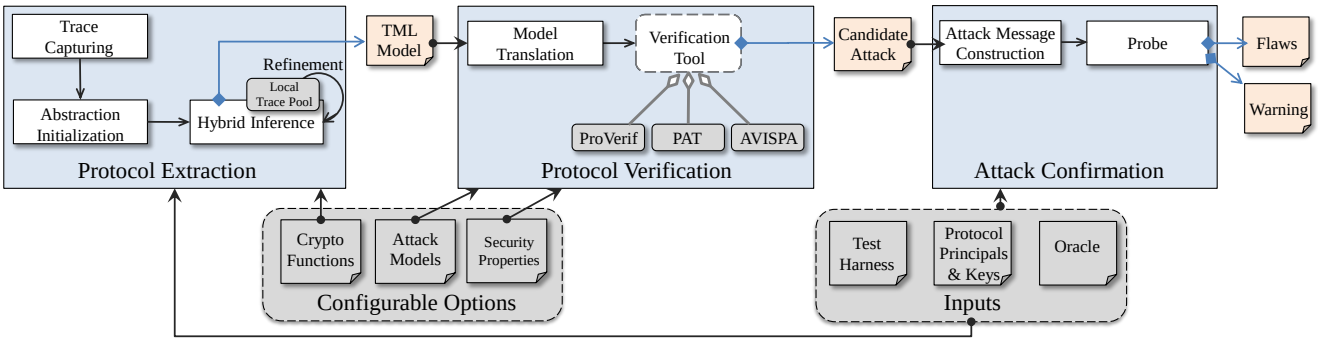


Figure 2: Overview of AUTHSCAN

Configurable Options. AUTHSCAN is designed to enable checking a variety of security properties under several different attacker models. Additionally, it is designed to incorporate domain knowledge that the security analyst is willing to provide to improve the precision. We next explain these configurable parameters of our system and defaults.

- **Attacker Models**. By default, AUTHSCAN checks for flaws against two standard attacker models: the network attacker [24] and the web attacker [15, 17]. However, it is possible to extend these models with new ones. For example, we can consider a filesystem attacker which steals authorization tokens stored on the client device. Such attacks have been found recently on the Android DropBox application [8].
- **Security Properties**. By default, AUTHSCAN checks for authentication of the inferred protocols. Checking authentication corresponds to two precise, formal definitions provided in previous work: *injective correspondences* [32] and *secrecy* [44]. Additional properties can be added to AUTHSCAN.
- **Cryptographic Functions Names**. AUTHSCAN needs to infer the functions which implement cryptographic primitives such as signature verification, hashes and so on, in the executed client-side JavaScript code (e.g. `verify` at line 27 in Figure 1-(c)). By default, AUTHSCAN performs this automatically. It has a built-in list of browser APIs (such as `Window.postMessage()`) and popular JavaScript libraries that provide such functions (such as Node.js [4] and Mozilla jwcrypto [9]). In addition, it has a small set of standardized cryptographic primitives. It can identify functions in the executed client-side code that mimic the behavior of these standardized functions using blackbox testing[2]. Security analysts can improve AUTHSCAN's precision and efficiency by providing additional names of JavaScript source code functions that compute cryptographic function terms.

[2]Alternative heavy-weight methods (e.g., [43]) to identify cryptographic functions using whitebox analyses are possible.

3 AUTHSCAN System Design

In this section, we present an overview of our techniques and introduce an intermediate language called TML to capture the full semantics of the extracted protocol.

3.1 Approach Overview

Figure 2 shows the internal design steps in our system. AUTHSCAN performs three high-level steps: *protocol extraction*, *protocol verification* and *attack confirmation*.

In the protocol extraction step, AUTHSCAN iteratively processes test cases one-by-one from its input test harness until the test harness is exhausted. For each test case, it records the network HTTP(S) traffic and client-side JavaScript code execution traces through a web browser. Using this information, AUTHSCAN generates an initial abstraction of the protocol specification. It then performs a refinement process to subsequently obtain more precise specifications[3]. In each refinement step, AUTHSCAN employs a hybrid inference technique which combines both whitebox program analysis on the JavaScript code (if available) and blackbox fuzzing. The refinement process stops if a fixpoint is reached (i.e., no new semantics can be inferred). Our protocol extraction techniques are detailed in Section 4.

At the end of the protocol extraction step, AUTHSCAN generates a protocol specification in an intermediate language called TML, which can capture the actions executed by each participant and the semantics of the data exchanged in the protocol execution. AUTHSCAN converts TML to applied pi-calculus, which is a widely-used specification language for security protocols. This protocol specification then can be automatically checked using

[3]By precise, we mean that each refinement contains more expressive semantics about actions performed by protocol participants and more relationships between data terms exchanged in the protocol.

off-the-shelf verification tools for various security properties, against different attackers. In this work, we use ProVerif [18] and PAT [38] as the verification tools because they can model an unbounded number of parallel sessions[4]. AUTHSCAN models various semantic restrictions, such as the same-origin policy, HTTP headers like `Referrer`, cookies, secure channels (HTTPS, `origin`-specified `postMessage`), and insecure channels (HTTP, unchecked `postMessages`), before querying off-the-shelf verification tools for precise reasoning, as detailed in [17]. Off-the-shelf verification tools verify these security properties and generate counterexamples which violate the properties. The counterexamples serve as unconfirmed or *candidate* attacks.

The last step of AUTHSCAN is attack confirmation step. In principle, our techniques can generate imprecise protocol specifications; therefore, some of the candidate attacks may not be true security flaws. AUTHSCAN can confirm HTTP attacks by converting counterexamples into HTTP network traffic, relaying them in a live setting and confirming true positives using the analyst-specified oracle. In the cases where AUTHSCAN does not know the attacker's knowledge set enough to generate confirmed attacks, it generates security warnings containing precise communication tokens that need to be manually reviewed by the security analyst.

3.2 Target Model Language

The semantics of our inferred authentication protocol is represented in an abstract language called Target Model Language (TML). TML serves as a bridge between protocol implementations and formal models supported by verification tools. It captures enough implementation-level details to check correctness, and at the same time, it can be translated into formal specifications that can be used as inputs to off-the-shelf security protocol verification tools.

We design TML based on the language proposed by Woo and Lam [44], referred as WL model in this work; we add new extensions which are necessary for our protocol inference. We explain the TML semantics in an intuitive way here to ease understanding; the terminology used (underlined) has precise semantics as defined in WL [44]. The TML representation of our running example is shown in Figure 3.

TML Syntax. TML represents an authentication protocol as a protocol schema. AUTHSCAN observes several concrete executions of a protocol, each of which is an instantiation of the protocol schema—for instance, our running example is an instantiation of our target protocol with two specific participants namely `idp.com` and `sp.com`. Formally, the protocol schema is a 2-tuple

[4]In this paper, we only use ProVerif to explain our idea. Bounded-state model checkers like AVISPA [10] can also be used but are not implemented as backends yet.

Initial Conditions

(I1) $\forall x, y : x$ has y
(I2) $\forall x, y : x$ has $key(x, y) \wedge y$ has $key(x, y)$
(I3) $\forall x, y : x$ has k_y
(I4) r has $sessionID_r \wedge p$ has $sessionID_r$
(I5) r has $CSRFToken_r \wedge p$ has $CSRFToken_r$
(I6) Z has $assoc(i, authtoken)$
(I7) i has $k_B \wedge r$ has k_B

SP_C(i) Protocol

SC1: BeginInit(j)
SC2: NewAssoc($\{p, i\}, assoc(j, spid)$)
SC3: Send(r,$\{[assoc(j, spid), next]\}_{k_B}$) // Step ①
SC4: Receive(r,$\{[M, N, \{[M, N]\}_{k^{-1}_{IDP\_S}}]\}_{k_B}$) //Step ④
SC5: Send(j,$[M, N]$) //Step ⑤
SC6: EndInit(j)

SP_S(j) Protocol

SS1: BeginRespond(i)
SS2: Receive(i,$[M, assoc(M, N)]$) //Step ⑤
SS3: EndRespond(i)

IDP_C(r) Protocol

IC1: Receive(i,$\{X, Y\}_{k_B}$) //Step ①
IC2: Send(p,$\{\{X, sessionID_r, CSRFToken_r\}\}_{key(r,p)}$) //Step ②
IC3: Receive(p,$\{\{M, N, P\}\}_{key(r,p)}$)//Step ③
IC4: Send(Y,$\{[M, N, P]\}_{k_B}$)//Step ④

IDP_S(p) Protocol

IS1: Receive(r,$\{\{assoc(T, U), sessionID_r, CSRFToken_r\}\}_{key(r,p)}$) //Step ②
IS2: NewAssoc($\{p, j\}, assoc(i, authtoken)$)
IS3: Send(r,$\{i, assoc(i, authtoken), \{[i, assoc(i, authtoken)]\}_{k^{-1}_{IDP\_S}}\}_{key(r,p)}$) //Step ③

Figure 3: The TML model of running example in Figure 1. *M, N, P, T* and *U* are variables. `I2` and the session keys in `IC2, IC3, IS1` and `IS3` model HTTPS communication. Cross domain restrictions by the browser's SOP are modeled as encryption using the key k_B (initialized in `I7`). j and p are identities of SP and IDP respectively, i.e., their domains. The behavior of Alice is modeled together on SP client side, thus i stands for Alice's `uEmail` which is Alice's identity. `sessionID` and `CSRFToken` have been inferred to be nonces (`I4` and `I5`). The `authToken` is inferred to be guessable (`I6`).

$(Init, ProSet)$. The $ProSet$ is a set of local protocols $\{P_1(X_1), P_2(X_2), \ldots P_i(X_i)\}$, where each local protocol P_i is executed by a protocol participant X_i. The local protocol specifies a sequence of actions that one participant can perform. The complete specification is characterized by a set of local protocols to be executed by multiple participants. X_i are variables in the schema that may be instantiated by concrete principals (such as `idp.com`) in a protocol instance. The second part of the protocol schema is a set of initial conditions *Init*, such as the initial knowledge set of each protocol participant prior to the start of the protocol. In the TML of our running example (Figure 3), we infer 7 initial conditions (`I1`–`I7`); we explain how these are derived during protocol extraction in Section 4.

Actions. In executing a local protocol, the participant executes a sequence of actions. Actions can be either communication actions, which send/receive messages with

Table 1: The Action Schema in IML

BeginInit(r)	NewNonce(n)
EndInit(r)	NewSecret($\mathbf{S}, n$)
BeginRespond(i)	Accept(N)
EndRespond(i)	NewKeyPair(k, k^{-1})
Send(p, M)	NewAssoc($\mathbf{S}$, $assoc(m_1, \ldots, m_n)$)
Receive(p, M)	

other participants, or internal actions which result in updating local state (or, formally the knowledge set) of that participant. These actions are listed in Table 1. The semantics of these actions are fairly intuitive as their names suggest, with the exception of NewAssoc which is explained later in this section. For example, BeginInit(r) states that an initiator of the protocol begins its role with a responder r. EndInit(r) states that the initiator ends the protocol with the responder r; BeginRespond(i) and EndRespond(i) are similarly defined with i being the initiator. Send(p, M) or Receive(p, M) means sending or receiving M to/from p, respectively. NewNonce(n) is the action of generating a nonce. NewKeyPair(k, k^{-1}) is the action of generating an asymmetric key pair, where k is the public key and k^{-1} is the private key. NewSecret(S, n) indicates the action of generating a secret, which is intended to be shared with (or distributed to) a set of principals S. Secrets can be data elements such as shared session keys. The secret distribution is only complete when all participants for whom the secret is intended have explicitly executed the Accept(N) action. Note that a participant following a local protocol only executes an action after it executes the preceding action state in the schema. As a result of executing certain actions, such as NewNonce and Accept, participants update their knowledge sets. Intuitively, a participant's knowledge set includes the data terms that it possesses or can compute, which can be used by the participant in communication messages. The attacker, denoted by the principal Z throughout this paper, is assumed to follow no local protocol and is free to execute any action at any step under the constraints of its knowledge set and the capability of the assumed attacker model.

Terms. We aim to recover as much semantics of the data exchanged and the internal state maintained for each participant as possible. To characterize these semantics, TML provides three kinds of terms: *constant* symbols, *function* symbols and *variable* symbols[5]. Constant symbols include names of principals (web origins), nonces, keys and integer constants. Function symbols include the encryption function $\{\cdot\}_{\_}$, the shared key function $key(\cdot, \cdot)$, the concatenation function $[\cdot, ..., \cdot]$, the set construction function $\{\cdot, ..., \cdot\}$ and the arithmetic functions ($+$,$-$,$/$,$*$, and modulo). The public key and private key of a principal P are denoted by k_P and k_P^{-1}, respectively. The symmetric key shared by principles P and Q is denoted by $key(P, Q)$. A term is ground if it only consists of constants and function symbols. Finally, variable symbols represent terms which are not ground.

We aim to recover the precise relationships between terms exchanged in the protocol. For example, our analysis infers that the value of `idpSign` is the signature of `uEmail` concatenated with `authToken`, as can be seen at line 27 of the running example—this translates to the statement labelled `IS3` in Figure 3. If a participant receives a data element whose precise semantics is not known by the receiver, we represent this data as a variable in TML. For example, consider `SC4` in Figure 3, we model the messages on the receiver side as variables M and N; the participant X_i executing local protocol P_i in the schema is a variable; the responder r in the BeginInit(r) is also a variable which will be instantiated with concrete values in an execution instance of a protocol schema.

New Extensions in TML. TML extends the WL model with three new extensions. The semantics of other operations are defined in the WL model; we discuss why these extensions are needed. The first extension is arithmetic function symbols. These operations are often utilized in generating sequence numbers from nonces, and, often lead to weak or predictable tokens. Our TML can capture such weak constructions and subject them to testing.

The second extension is a function symbol called *association relation*, which is written as $assoc(m_1, \ldots, m_n)$ to associate n variables, m_1 to m_n. Association relation is necessary because while reconstructing the semantics from implementations, we sometimes cannot infer the exact relation between the terms even though we can infer that they are related. For instance, in the running example, we can infer that `authToken` (line 14, Figure 2) does not change during the sessions of the same user, and hence it is related to the user's identity, but the exact semantic relation is unknown. In this scenario, AUTHSCAN generates an association $assoc(i, authtoken)$ to indicate that the two terms are related as a key-value pair, but without the exact relation known.

The third extension we introduce in TML is an internal action called NewAssoc($\mathbf{S}, assoc(m_1, \ldots, m_n)$). This action means that the association $assoc(m_1, \ldots, m_n)$ is known or becomes shared among the principals listed in the set $\mathbf{S}$. To see why the sharing among $\mathbf{S}$ is needed, consider the following scenario. Principals P and Q possess a mutual shared secret k, that is known prior to the execution. P sends Q a message m in the client browser, both participants send m back to their backend servers, and their

[5]This typesetting is kept consistent with the WL model paper [44]. The constant symbols are typeset in Sans Serif font, the adversary is referred to as the principal Z and the universe of principals is the set SYS. Lower case variables stand for terms that are constant symbols, while upper case variables stand for arbitrary terms.

servers later respond with entity $\{m\}_k$ in subsequent HTTP messages observed in the browser. AUTHSCAN observes that P and Q compute the same term from m in the code hidden on their servers, but it cannot infer the exact relation between $\{m\}_k$ and m because it does not know that k is a pre-exchanged shared secret. Under such situations, AUTHSCAN introduces a NewAssoc action in the inferred protocol schema to specify that this association is known to both P and Q. The step `SC2` in Figure 3 shows how this relation is captured at TML.

We define the semantics for these extensions, which extends the original semantic model of the WL model in the following way. We introduce an *association table* for each principal to record the principal's knowledge of associations. When a principal executes NewAssoc($\mathbf{S}, assoc(m_1, \ldots, m_n)$), the $assoc(m_1, \ldots, m_n)$ is added into the association table of each principal in $\mathbf{S}$. Note that the attacker (i.e., Z) is not allowed to update the association table. When a principal receives an association, it checks implicitly if the association is stored in its table.

Assumptions in TML. We make the following assumptions in TML.

- **Correct Cryptographic Algorithms.** TML assumes that the cryptographic algorithms used in the protocol are ideal. We do not aim to detect vulnerabilities in the implementations of the cryptographic primitives.
- **Distinct Secret Keys and Nonces.** TML assumes the encryption/decryption keys are kept secret prior to the protocol, and are distinct (i.e., cannot be guessed).
- **Knowledge of Principals.** We make the assumption on the knowledge of the principals: Each principal knows the identifiers or names of other principals (represented as (I1) in Figure 3). This assumes that the DNS infrastructure has no vulnerability.

4 Protocol Extraction Techniques

In this section, we give the details of the proposed hybrid inference approach to address the challenges in Section 2.2.

4.1 Overview of Protocol Extraction

Our protocol extraction technique operates on the input test harness, one test case at a time. Figure 4 shows an overview of the protocol extraction process. As the first step, the *abstraction initialization* component in our system creates an initial abstraction of the protocol from the first test case in the test harness. It takes HTTP traces (captured by our trace capturing component shown in Figure 2) and the initial knowledge provided by the analyst as inputs. The initial abstraction of the inferred protocol is in the form of a TML protocol schema $(Init, ProSet)$. By utilizing the test cases from the test harness one-by-one, AUTHSCAN iteratively refines the abstract protocol using our hybrid inference technique discussed in this section. During each iteration of the hybrid inference, AUTHSCAN gradually refines the semantics of terms and actions of the protocol schema until no new semantics can be discovered.

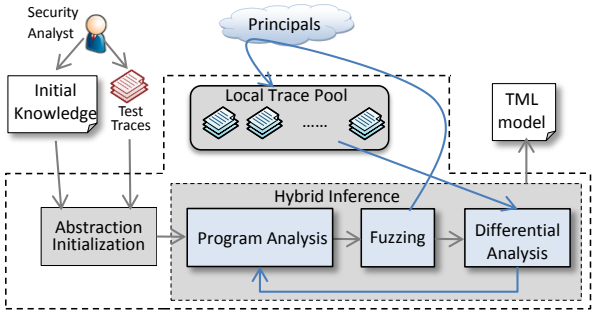


Figure 4: AUTHSCAN's protocol extraction process

Algorithm 1 Abstraction Refinement Algorithm

Require: $InitK$: initial knowledge, t: test trace
Ensure: PS: protocol schema
1: $(Init, ProSet) \leftarrow absInit(t, InitK)$;
2: $ProSet_{old} \leftarrow null$;
3: $trPool$: a trace list, initially empty
4: **while** $ProSet \neq ProSet_{old}$ **do**
5: $\quad ProSet_{old} \leftarrow ProSet$;
6: $\quad ProSet \leftarrow JSAnalysis(t, ProSet)$;
7: $\quad (ProSet, T) \leftarrow Blackbox(t, ProSet, InitK, trPool)$;
8: $\quad trPool.add(T)$;
9: **end while**
10: **return** $(Init, ProSet)$;

4.2 Protocol Refinement Algorithm

The protocol refinement algorithm is shown in Algorithm 1. The inputs of the algorithm are the initial knowledge $InitK$ (i.e., the test harness, protocol participants & public keys of participants and oracle, outlined in Section 2.3), and a trace t generated from one test case. A trace is a sequence of messages $(a_0, a_1, ..., a_n)$, where a_i represents either an HTTP(S) request, response (which may contain JavaScript programs), or a cross-domain communication message over `postMessage`. We refer to all data exchanged in the trace as *HTTP data*, which includes HTTP parameters, cookies, `postMessage` data, HTML form data, JSON data, and so on. AUTHSCAN's trace capturing step identifies the HTTP(S) request/response pairs from the trace. The output of the algorithm is one inferred protocol schema.

Our refinement algorithm (Algorithm 1) has two steps: abstraction initialization (line 1) and refinement process (line 4-9). The $absInit$ method (line 1) returns an abstract protocol schema $(Init, ProSet)$. $Init$ is a set of predicates, which stands for the initial knowledge of the

principals. Some of these are derived from the assumptions of TML (outlined in Section 3.2), e.g., $I1 - I3$ shown in Figure 3. Other TML terms model the communication channels that are used in the protocol. For example, to model the HTTPS channels and cross-domain communication channels, we internally introduce symmetric keys ($I6$ in Figure 3), as we explain in Section 5.2. For every message a in test trace t, if the sender or the receiver of a is not contained in $ProSet$, $absInit$ inserts a new local protocol into the $ProSet$. Then, $absInit$ adds two communication actions (Send and Receive) into the sender's and receiver's protocol, respectively. In addition, $absInit$ can identify some constant terms in the HTTP data, such as the domains of principals, user accounts and public keys of web sites available as the security analyst's inputs to AUTHSCAN. AUTHSCAN identifies them by matching the value of HTTP data with the values in the analyst's inputs. For example, i, r and k_{IDP_S} are identified in this way; they stand for the identity of SP, the identity of IDP and the public key of IDP, respectively. At the end of this step, other HTTP data, which cannot be inferred here, are represented as variable terms whose semantics are inferred in the refinement process explained next. The Begin* and End* events are also inserted into the local protocols indicating the SP's client and server.

In the refinement step (line 5-8), AUTHSCAN refines the initial abstraction by utilizing more test cases. This step combines whitebox symbolic analysis ($JSAnalysis$ at line 6) and a blackbox analysis ($Blackbox$ at line 7).

Whitebox Program Analysis. The $JSAnalysis$ procedure uses dynamic symbolic analysis (at line 6) to infer the function terms and the internal actions of the principals. Dynamic symbolic analysis (similar to previous work [35]) is used to obtain symbolic formulae which capture the relations among the HTTP data. These symbolic formulae are over the theory of TML terms, which include arithmetic operations, concatenation function, cryptographic operations and uninterpreted functions. We introduce uninterpreted functions to model semantics unknown function calls, such as calls to browser APIs or JavaScript functions which have many arithmetic and bitwise operations characteristic of cryptographic operations. For the code fragment marked **B** in our running example (Figure 1), if the input value for the variable `event.data` is a string "`u&t&s`", the following symbolic formulae are generated by this step:

(1) $uEmail := u$; (2) $authToken := t$;
(3) $idpSign := s$; (4) $data := [u, t]$;
(5) $idpPubKey := loadPubKey()$;
(6) $verify([u, t], s, idpPubKey)$;
(7) $message := [u, t]$;
(8) $request := \$.ajax(login, [u, t])$;

To precisely identify cryptographic function terms in the symbolic formulae, AUTHSCAN needs to identify JavaScript functions implementing cryptographic signature, encryption, random number generation, public key fetching functions and so on. From the above symbolic formulae example, $JSAnalysis$ can identify that `idpSign` is the TML term $\{[uEmail, authToken]\}_{k_{IDP}^{-1}}$, once AUTHSCAN knows that the semantics of the JavaScript procedure `verify` $(data, sig, key)$. By default, AUTHSCAN identifies these functions based on its built-in list of browser APIs and JavaScript libraries that provide such functions [4]. AUTHSCAN tries to concretely match the semantics of all symbolic terms identified as uninterpreted functions in the symbolic formulae to one of known cryptographic functions in its built-in list. For example, AUTHSCAN can test `verify` with the same inputs as the standard RSA signature verification function from its built-in list and compare the outputs. Security analysts can also provide annotations for source code functions to identify custom implementations of standard cryptographic primitives, in case the default list is not sufficient. In this way, several variables are replaced with newly inferred TML terms in this step. For an uninterpreted function whose semantics cannot be inferred in this step, AUTHSCAN uses an $assoc$ to represent it. The $assoc$ associates the output of the function with the inputs.

Based on the extracted symbolic formulae, $JSAnalysis$ infers the function terms and some internal actions in local protocols. For example, if an HTTP data is identified as a session key, AUTHSCAN treates the principal which first sends it in the communication as generator of this session key. AUTHSCAN infers that this principal has performed a NewSecret action and the principals which receive it have performed Accept actions. If a principal invokes an asymmetric key pair generation function, AUTHSCAN adds a NewKeyPair action to the principal's protocol.

Blackbox Differential Fuzzing Analysis. The blackbox analysis (at line 7) further refines the output of the whitebox analysis by trying to infer more TML terms and actions while treating the participant implementations as a blackbox. Our blackbox differential fuzzing analysis takes as input the trace t, the refined abstraction after whitebox analysis, and the initial knowledge $InitK$. The first substep in blackbox fuzzing is to remove certain redundant data to make blackbox testing more efficient. Next, the blackbox inference algorithm infers TML terms in two ways: for some terms, it generates "probe" messages and compares the outputs, whereas for other terms, it merely makes the inference based on the observed traces without generating new probes. We describe the redundant data elimination, probe-based inference and non-probe-based inference substep separately. In each iteration of the blackbox fuzzing step, AUTHSCAN internally generates new traces and keeps them in a local trace pool ($trPool$ in Algorithm 1). These traces are not fed back to the initial test harness, and are

used only during the blackbox and whitebox steps.

Eliminating Redundant Data. The goal of this step is to identify HTTP data that do not contribute towards the authentication protocol. In this step, we check each HTTP data element by generating a probe message with this element removed. If the probe message results in a successful authentication, we remove the element and all of its occurrences in previous messages. AUTHSCAN performs this operation iteratively for each request/response pairs starting from the last pair and proceeding backwards in t.

Probe-based Inference. The main idea of this fuzzing step is to mutate or remove the HTTP data in the request messages of t, while keeping others unchanged. These modified "probe" messages are sent to the protocol participants and their responses are compared for differences. In addition, to prevent the explosion of number of HTTP traces, we capture at most three traces for each test user account and at most 10 test user accounts for each web site. AUTHSCAN identifies the semantics of several types of HTTP data: URLs, HTTP parameters, web addresses, JSON data, JSON Web tokens, and web cookies. To do this identification, it uses simple pattern matching rules over the values of the data. For instance, a string which has sub-strings separated by "&", with each segment as a key-value pair separated by a "=", is treated as an HTTP parameter list. Similar syntactic properties are used for common web objects such as JSONs, JWT, cookies and so on. Once the HTTP data type is inferred, AUTHSCAN makes use of the type information to speed up the fuzzing process. For example, if AUTHSCAN infers that a string is an HTTP parameter-value list, it mutates each key value pair in this string separately. Similarly, if AUTHSCAN infers that a string represents a user identity (like usernames) or a web address, it mutates the value of this HTTP data into another user's ID or another web address, instead of trying random modifications. AUTHSCAN also incorporates simple pattern-matching rules to identify if values are encoded using common encoding methods such as URLEncode/URLDecode, Base64-encode, HexEncode, HTMLEncode and JavaScript string literal encode, based on the use of special characters. For an HTTP data with completely unknown semantics, AUTHSCAN uses pattern-matching techniques to label it as one of primitive types (Integer, Bool, or String).

Once the basic types are identified, AUTHSCAN then infers the TML terms and actions. From the traces in the local trace pool, AUTHSCAN attempts to first identify arithmetic function terms, which in turn enables the modeling of weak or guessable tokens. For Integer- or String- typed value of an HTTP data parameter that change across sessions, AUTHSCAN uses the following mechanism to check if it is generated using a predictable arithmetic function. Given such a string value (say str), AUTHSCAN first conducts a substring matching between its instances across various traces and extracts the parts that are not common between these instances. AUTHSCAN then checks if these values form simple arithmetic sequences adding or subtracting a constant. If the function is identified, AUTHSCAN treats it as a guessable token, and confirms it by predicating its value and probing the server (discussed in Section 5.3). We plan to integrate more powerful off-the-shelf tools, such as Wolfram Alpha, which take such value sequences as inputs and output a closed form arithmetic expression to match it [11]. AUTHSCAN also marks any data value which is too short ($L \leq 4$ characters by default and configurable) as guessable *short-length* tokens, as these values may be subject to exhaustive search. For example, in the case where $L = 4$, the search space is less than 2 million ($(10 + 26)^4$), assuming that the term only consists of case-insensitive alpha-numeric characters; AUTHSCAN presently does not actually generate these probes but models such values as attacker's knowledge (as detailed in Section 5.2), and generates security warnings.

Next, AUTHSCAN infers two kinds of associations using techniques similar to those proposed by Wang et. al. [42]. One kind of association is among HTTP data. AUTHSCAN replaces the value of an HTTP data x in message a_i, while keeping the rest unchanged. Then it sends this "probe" message and compares the response message. If HTTP response $\vec{y}$ changes, AUTHSCAN introduces an $assoc(x, \vec{y})$. Other kinds of association relations are between HTTP data and a web principal or users. Similarly, AUTHSCAN identifies these associations by using differential analysis on multiple traces. The HTTP data which remain constant among the same user's multiple sessions are inferred to be associated to the user; those remaining constant among different users' sessions are inferred to be associated with a web principal (such as the SP or IDP). All remaining HTTP data that change in all such probes are inferred to be nonces (NewNonce), such as session IDs.

Identifying Association Principals. The **S** in NewAssoc(**S**,...) stands for the principals who share the knowledge of the association terms. AUTHSCAN identifies these principals by observing which terms in an $assoc$ appear in the responses from the protocol participants. Then, it probes these participants by replacing the associated terms with random values. If a principal rejects the fuzzing message, we infer that it knows how to compute the relationship, and add a NewAssoc with these participants in **S**.

Non-Probe Based Inference. The non-probe based inference infers three kinds of function symbols: cryptographic functions, set functions and concatenation functions. AUTHSCAN employs brute-force search to identify cryptographic functions. It takes every combination of all HTTP data elements and checks if they can be used as inputs to a standard cryptographic primitive to produce an-

other data element. We bound the function nesting depth of terms to be less than 5. In our experiments, we find that this bound is reasonable since all our analyzed protocols do not use no more than 4 levels of nesting cryptographic constructions. This search strategy has been sufficient in practice for our experiments on real-world protocols. For example, as discussed in our BrowserID case study (Section 6), AUTHSCAN successfully identities that one HTTP data element is signed by the IDP, and that the signed elements are the ID and the user's public key. AUTHSCAN identifies the concatenation functions by using a substring search over all combinations of HTTP data elements. For the set construction functions, if a single message contains multiple data, AUTHSCAN assigns them to a set.

5 Protocol Analysis & Attack Confirmation

After extracting a TML model, AUTHSCAN translates it into applied pi-calculus, which is taken as input to ProVerif [18] to check security properties against attack models. Due to space constraints, we leave the details of this process to Appendix C; and in this section, we discuss the security properties, attacker models and how candidate attacks are checked to confirm security flaws.

5.1 Security Properties

By default, AUTHSCAN checks the correctness of two essential security properties in its applied pi-calculus version, *authentication* of an authentication protocol [44] and *secrecy* of credential tokens. A protocol achieves authentication if each principal is sure about the identity of the principal whom it is communicating with. Authentication is checked using *injective correspondence* ($\rightsquigarrow$, or *injective agreement*) [19, 20, 32, 44], which can check whether two local protocols are executing in "lock-step" fashion, i.e., whether there is an injective mapping between the execution of two participant's protocols. For instance, in our running example, whenever finishing executing EndRespond(i), *SP_S* believes that *SP_C* has executed the protocol with him; thus, to guarantee authentication, *SP_C* must have executed BeginInit(j), i.e., EndRespond(i) $\rightsquigarrow$ BeginInit(j) (`inj-event(EndRespond(i))==>inj-event(BeginInit(j))` in applied pi-calculus). Authentication is violated if *SP_S* believes *SP_C* has executed the protocol with him, but actually it is Z who has.

Additionally, an authentication protocol may introduce some credentials and thus *secrecy* of them needs to be guaranteed. Secrecy is defined as querying a term from the attacker Z's knowledge set [44]. The secrecy of a term a is specified as Z has a (`query attacker(a)` in applied pi-calculus), which queries whether a is derivable by Z after the execution of the authentication protocols. If Z has a after the protocol, the protocol fails to guarantee the secrecy of a. By default, AUTHSCAN checks the secrecy of terms used for authentication (such as the `sessionID` in the running example); the attack analyst can add more queries to check the secrecy of other terms, for example, credentials for resource access (such as OAuth token in OAuth 2.0). For long-lived tokens, AUTHSCAN adds them to Z's knowledge set before querying ProVerif. In general, Z may know a long-lived token's value (through external knowledge) even if it is not sent on a public channel; AUTHSCAN conservatively models this scenario and raises a security warning to alert the analyst. For guessable tokens, AUTHSCAN adds the outputs of the arithmetic operations to Z's knowledge set. In the attack confirmation step, these guessable values are computed and used as we detail in Section 5.3.

5.2 Attacker Models

In this work, we consider two different attacker models, namely the network attacker [24] and the web attacker [15]. Previous work (e.g., [17]) has shown that these attackers can be captured in ProVerif. Hence, we ignore the detailed modeling and just give an overview in this section. For example, attacker model in the running example is demonstrated in Appendix C. Note that both the attacker models are checked individually in AUTHSCAN, since ProVerif terminates after finding a counterexample.

Network Attacker. We model the network attacker using the Dolev-Yao model [24], that is, an active network attacker is able to eavesdrop all messages and control the contents of unencrypted messages in the public network under the constraints of cryptographic primitives. In TML, we model HTTPS by assuming that the SSL certificate checking and handshake are complete before the protocol starts; we model the session key between the two communicating principals x and y with a key function $key(x, y)$ (`I2` in Figure 3). In applied pi-calculus, we model HTTPS using private channels, which are neither readable nor writable by the attacker (shown in Appendix C). Note that modeling the HTTP network attacker is available from ProVerif directly.

Web Attacker. We also reuse web attacker models described in prior work [15, 17]. These models include modeling the same-origin restrictions; for example, the fact that client-side SP code cannot intercept IDP server's messages is implied in the applied pi-calculus semantics that the local variables of a process are inaccessible by another process. We model HTTP headers like `Referrer` which correspond to the client-side code sending its identity in the messages; of course, if the header is not checked by the server, it will not be inferred in our specification as it is removed as a redundant element. We also model the semantics of `postMessage` by encrypting all messages transmitted through `postMessage` with a key (k_B in `IC4` and `SC4`, Figure 3). If AUTHSCAN finds (by whitebox analysis) that the receiver or sender `origin` fields are not checked, it casts k_B to the attacker such that the attacker is able

to read and write the `postMessage` channel. The anti-CSRF tokens are not needed to be explicitly modeled in the attacker model as they are observed in the HTTP network messages and are inferred to be nonces if they are relevant to the protocol (`I4` and `I5` in Figure 3). We assume that the attacker has the ability to redirect the user agent to a malicious web site. We do not model web attackers with the ability to perform Cross-Site Scripting (XSS) attacks and complex social-engineering attacks in this work.

5.3 Candidate Attack Confirmation

AUTHSCAN confirms candidate attacks generated by ProVerif in this step. If a protocol fails to satisfy the security properties, ProVerif generates a counterexample, which consists of the attacker's actions, the attacker's input/output and details the terms computed by Z at each step using it's knowledge set at that step. AUTHSCAN re-constructs the candidate attack probe from this information. For all terms computed at each step, AUTHSCAN substitutes the concrete values for these terms. For guessable tokens that are computed from arithmetic functions, AUTHSCAN evaluates the function to calculate the next concrete value. For short-size guessable tokens, AUTHSCAN only raises a security warning. To map symbols and variables in ProVerif counterexamples to concrete values observed in the HTTP traces, AUTHSCAN maintains the mapping between the original HTTP messages and the protocol statement generated during the protocol extraction. Thus, AUTHSCAN maps back a ProVerif action sequence and terms in the ProVerif counterexample to the ProVerif input, which inturn is mapped to the raw HTTP message. Once the messages are constructed, AUTHSCAN replays the candidate attack probe. During this process, it queries the oracle provided by the analyst to check whether the attack is successful.

Currently, AUTHSCAN automates confirmation of attacks over HTTP, over `postmessage` and via a web attacker-controlled `iframe`. In cases which AUTHSCAN cannot confirm with concrete attack instances, it reports security warnings containing the communicated data it suspects. Such cases include the use of long-lived token in authentication, secrecy of which is not known in the inferred protocol but conservatively modeled as discussed in Section 5.2, and the use of guessable short-length tokens.

6 Evaluation

We have built an implementation of AUTHSCAN in approximately 5K lines of C# code, and 3K lines of JavaScript code. The HTTP trace recording and blackbox fuzzing functionalities are implemented in a Firefox add-on. The JavaScript trace extraction is implemented by instrumenting the web browser to generate execution traces in a format similar to JASIL [36]. We developed our own implementation of dynamic symbolic analysis for extracting the TML terms from the execution traces.

6.1 Evaluation Subjects

To estimate the effectiveness of AUTHSCAN on real-world protocols, we test several implementations of popular SSO protocols and standalone web sites that implement their custom authentication logic. The inferred protocols are presented in Appendix B.2. Our results are summarized in Table 2.

BrowerID. BrowserID [2] is an SSO service proposed by Mozilla, which is used by several Mozilla-based services such as BugZilla and MDN, as well as some other service providers. We test three different SP implementations of BrowserID. Although BrowserID is open-source, most of protocols do not provide the detailed implementation on the server-side. To account for this, we only take into consideration the client-side JavaScript code and HTTP messages to make our analysis approach more general. AUTHSCAN manages to infer the general protocol specification from these three implementations, finding only one crucial difference across the implementations (explained in Section 6.2).

Facebook Connect. Facebook Connect [3] is one of the most widely used incarnations of the OAuth 2.0 published by Facebook. We test two SP web sites using this protocol. The experiments are conducted on the basis of client-side JavaScript code and HTTP messages. AUTHSCAN infers the general protocol specification successfully.

Windows Live ID. Windows Live Messenger Connect [6] is another SSO protocol derived from the general OAuth 2.0 specification. We test its implementation using the Sina Weibo service—a China-based web site similar to Twitter and has over 300 million users. AUTHSCAN successfully extracts the protocol from this implementation; we skip the protocol diagram (which is similar to Facebook Connect) for the sake of space.

Standalone Web Sites. We also test two standalone sites, where users share deeply personal information, both of which have from hundreds of thousands to millions of users and utilize custom authorization mechanisms. AUTHSCAN uncovers the custom authentication protocol for both sites.

6.2 Protocol Analysis and Vulnerabilities

We test AUTHSCAN on 8 implementations (as shown in Table 2). We successfully find 7 security vulnerabilities, all of which we have responsibly disclosed to the developers of the web sites. For the sake of space, we leave the details on how AUTHSCAN extracts protocol specification to Appendix B.1; and in this section, we briefly present the found vulnerabilities in the protocol implementations.

Setup. In our experiment, the input and configuration to AUTHSCAN include:

- *Test harness.* The security analyst is required to input two pre-registered user accounts (for example, `email` and `password` in BrowserID), except for the Iyer-Matrimony case in which five are needed.

Table 2: Statistics in our experiments

Column 2: ratio of messages filtered out by AUTHSCAN w.r.t. the total number of messages occurred in the protocol; Column 3: ratio of parameters filtered out by AUTHSCAN w.r.t. the total number of parameters used in the messages; Column 4: total execution time of AUTHSCAN; Column 5: verification time of running ProVerif *without* and *with* filtering of the messages or HTTP data, under the network attacker, where "-" means nontermination in verification; Column 6: number of rounds; Column 7: number of bugs found in each web site (with repeats); there are 7 distinct (without over-counting) vulnerabilities.

Web Sites	% Redundant Msgs (Total Msgs)	% Redundant Elems (Total Elems)	Time(s)	Verification Time (s) WO (W Filter) Filter	Fuzzing Rounds	#Bugs
myfavoritebeer.com	88% (80)	50% (12)	113	204/3.0	20	2
openphoto.me	82% (93)	75% (24)	72	726/3.0	22	2
developer.mozilla.org	87% (127)	74% (23)	96	-/3.0	28	0
ebayclassifieds.com	72% (58)	57% (152)	127[a]	-/58.7	107	2
familybuilder.com	97% (290)	51% (144)	110[a]	-/58.7	77	1
weibo.com	97% (176)	98% (52)	30	0.36/0.03	78	1
iyermatrimony.com	98% (120)	67% (9)	5.33	1.14/0.04	510	1
meetingmillionaires.com	96% (54)	0% (5)	4.72	1.05/0.04	30	1

[a] The period that AUTHSCAN halts until Facebook allows to resume fuzzing is not taken into account.

- *Protocol principals & public keys.* For the SSO implementation (including BrowserID, Facebook Connect and Windows Live ID), the analyst needs to indicate domains of IDP and SP (for example, in BrowserID case, `persona.org` and `myfavoritebeer.org`, respectively). For the standalone web sites, the analyst needs to indicate the domains of the tested sites. In both cases, the public keys of the participants need to be provided if HTTPS is used in the implementation.
- *Oracle.* The analyst needs to provide an indication to represent the successful authentication. In our experiments, we provide unique strings on the response webpage from the server such as "welcome user" to identify if the authentication succeeds.
- *Cryptographic functions.* We manually annotate the cryptographic functions in the Crypto library of Node.js [4], for AUTHSCAN to identify the cryptographic functions. We also annotate the functions in Mozilla jwcrypto [9], which is used in the implementation of BrowserID. AUTHSCAN automatically infers cryptographic operations using its default method in all other case studies.

For all cases, AUTHSCAN checks the authentication of the protocol and secrecy of the terms used for authentication (such as the *assertion* in BrowserID, which is discussed later in this section). These properties are checked against the network attacker as well as the web attacker.

Replay Attack in BrowserID. In two tested implementations of BrowserID, which use `persona.org` as IDP, AUTHSCAN identifies and generates a confirmed *replay attack* in the network attacker model. AUTHSCAN generates an attack HTTP trace in which a malicious user logs into the SP by replaying the token named *assertion* (message (7) in Figure 6), without providing login credentials to the IDP. The flaw leading to this attack is that the assertion is sent through an insecure channel (HTTP) and it does not contain any session-specific nonce. We recorded a video to demonstrate that the attack works and proposed to add a nonce in the signature to solve this problem [1]. We have notified Mozilla about our finding and Mozilla acknowledged the security flaw.

CSRF Attack in BrowserID. AUTHSCAN identifies and confirms a replay attack in the web attacker model. AUTHSCAN reports this attack on two of the BrowserID implementations, other than the one from `developer.mozilla.org`. We have responsibly notified the vendors of these vulnerable implementations. After manual analysis of the inferred protocols, we find one crucial difference between the vulnerable implementations from the `developer.mozilla.org` implementation. In the latter, SP client sends two anti-CSRF tokens (`csrfmiddlewaretoken` and `next` which are inferred as nonces) in step 7 (Figure 6), but these are absent from the protocol schema of the vulnerable SPs implementation, permitting a CSRF attack. AUTHSCAN reports that a malicious web site can send an HTTP POST request to the vulnerable SPs, which do not check the Referrer fields. Using this knowledge, we craft a script which can be used by the attacker to modify the content on the web pages without Alice's approval. The attack script is listed in Appendix D.

Secret Token Leak in Facebook Connect. By following a similar procedure as illustrated in the case of BrowserID, AUTHSCAN finds one confirmed flaw in the implementation of Facebook Connect, and another one in the usage of Facebook Connect by one out of the two SPs we tested. Both attacks leak secret tokens in the network attacker model. In this case, we report that automatic fuzzing was initially difficult because Facebook blocks login failure for a test username/password after 10 attempts. For this, we manually skipped fuzzing the initial login request to the IDP, but tested the remaining protocol with the SPs.

In the implementation of Facebook Connect, most of the communications are through HTTPS to prevent network at-

tackers from stealing the authorization tokens. However, AUTHSCAN reports that the message at step 4 of Figure 6-(b) is readable to the network attackers because they are transmitted through a non-HTTPS channel, so two credentials `c_user` and `xs` can be obtained by the attacker. Thus, the protocol is subject to a replay attack similar to the one in BrowserID. After our experiments, we discover that a similar attack against the previous version of Facebook Connect has been reported by Miculan *et al.* recently [33]. We conducted our tests in the end of April 2012; Facebook fixed this flaw in early May 2012 before we were able to notify them. In Facebook's latest implementation[6], the communication in this step is protected with HTTPS. We provide the HTTP/HTTPS messages captured during the execution of the old version to facilitate further analysis, which can be downloaded from [1]. AUTHSCAN finds the other flaw leading to replay attack when an SP called EbayClassifieds uses the Facebook Connect. After completing the Facebook Connect, the SP sends the user credentials which can be used to fetch session cookies. However, the credential is also sent through a non-HTTPS channel.

Non-secret Token in Using Windows Live ID. We tested AUTHSCAN on the authentication mechanism of Sina Weibo, a web site with 300 million users. It uses Windows Live ID to authenticate users. AUTHSCAN initially reported a security warning claiming that a long-lived token (non-nonce value) is used to authenticate the user. We subsequently manually investigated this warning, and found that the long-lived token (named `msn_cid`) reported by AUTHSCAN is known publicly. For example, it can be obtained from various sources such as straight from the MSN user profile page (`https://profile.live.com/cid-xxxx`). When we added this token to the attacker's knowledge set and re-ran the experiment, AUTHSCAN was able to automatically generate an attack trace.

This flow occurs after a user completes the authentication with Windows Live ID, which demonstrates that AUTHSCAN is useful for finding simple, but severe logic flaws beyond the initial SSO authentication token exchange. Note that manually finding these attacks is not easy; AUTHSCAN eliminated 18 redundant cookies with differential fuzzing. The final HTTP packet which is sent from user to Weibo web site for authentication, as constructed by AUTHSCAN, sets the `msn_cid` value to the publicly known value as shown below.

```
GET /msn/bind.php HTTP/1.1
Host: www.weibo.com
Connection: keep-alive
Cookie: msn_cid=xxxx
```

This vulnerability impacts all Weibo users who have ever logged in Weibo through Windows Live Messenger. We have reported this security flaw to Sina Weibo. The security department of Sina R&D has confirmed the exploit and posted us a gift for our contribution [1].

[6]https://s-static.ak.facebook.com/connect/xd_arbiter.php?version=9

Guessable Token in Standalone Sites. AUTHSCAN detects one severe vulnerability in each of the two standalone web sites: IyerMatrimony and MeetingMillionaries. Both of them have a significant number of registered users, 220,000 and 1,275,000, respectively. The vulnerability shows that both of these two web sites authenticate users by some guessable token. Exploiting these vulnerabilities, the attacker can log into others' accounts and get full privilege of the victim users.

In the case of IyerMatrimony, after eliminating 7 redundant HTTP parameters with differential fuzzing, AUTHSCAN gets the following packet which can be used for a successful authentication.

```
http://www.iyermatrimony.com/login/
    intermediatelogin.php?sde=U1ZsU01UZ3dOVE01
    &sds=QdR.j/ZJEX./A&sdss=Tf/GpQpvtzuEs
```

Through differential fuzzing, AUTHSCAN finds that `sds` and `sdss` keep constant among different accounts' multiple login sessions; for an individual account, the `sde` remains the same in its multiple sessions. Among the test accounts, AUTHSCAN finds that the 14-character prefix of `sde` remains constant and only the 2-character postfix is incremented by one across accounts whose IDs are consecutive numbers. AUTHSCAN confirms this flaw by predicting the value of `sde` for our testing accounts and successfully logging into the account.

In the MeetingMillionaries case study, AUTHSCAN generates a security warning about a short-length token used for authentication. We manually confirmed that this warning is a security flaw and notified the developers. In this site, a user can access his account information (including password stored in plain text) by visiting the following URL.

```
http://app.icontact.com/icp/mmail-mprofile.pl?
r=36958596&l=2601&s=21DS&m=318326&c=752641
```

AUTHSCAN finds l, m and c are constant among different users' sessions and r is associated with the user account. s is the only credential but due to its short length (4 characters), AUTHSCAN raises a warning of guessable token. Upon our manual investigation, we find that s is an alphanumeric string. We believe that automating attack generation for such tokens may be possible in the future; we tested that AUTHSCAN can send about 500 requests to the server within one minute. With such capability, it would take an enhanced implementation of AUTHSCAN at most 56 hours to guess the right s.

6.3 Efficiency & Running Time

Running Time. The total analysis time for most cases is less than 2 minutes, and can be as low as 5 seconds. The verification time for ProVerif is within 1 minute in our case studies. It shows that the security-relevant parts of the protocols generated are usually small. We find that additional source code results in the reduced number of iterations in our blackbox fuzzing step. For example, in BrowserID,

the client-side code is available, therefore, the number of fuzzing iterations is smaller (20-30 rounds) than other SSO protocols (30-500 rounds as shown in the sixth column, Table 2). Our data shows that AUTHSCAN's protocol extraction step is sufficient to find flaws even when much of the protocol implementation is unavailable as shown in the Facebook case.

Redundant Data Reduction. When querying off-the-shelf verification tools like ProVerif, it is important to remove redundant terms for better scalability. As shown in Table 2, AUTHSCAN finds that the majority of the messages (more than 80%) and HTTP parameters (more than 50%) are irrelevant to the protocol and AUTHSCAN can successfully filter them out. This shows that an automatic tool is helpful in constructing the models from the complicated implementation details. Furthermore, this reduction helps greatly in reducing the verification time. For BrowserID, ProVerif does not terminate within one hour if we naively retain all terms exchanged in the communication. In summary, we find the AUTHSCAN has promising scalability for real-world security protocol implementations.

7 Related Work

Protocol Specification & Verification. Security protocol verification has been well studied in the literature. Many logics and calculi have been proposed to formally specify the security protocols and security properties, such as BAN logic [13, 21], WL model [44], Spi-calculus [12]. A number of automatic verification tools have been developed and used to check the correctness of the security protocols, such as Athena [37], ProVerif [18], Murphi [34] and AVISPA [10]. These works focus on verifying the high-level specifications of the security protocols. However, our approach focuses on how to extract the high-level protocol specification from the implementations.

Protocol Extraction. Works on automatically extracting models from the protocol implementations are most related to this work. Lie *et al.* [30] have proposed a method to automatically extract specifications from the protocol code. The model is extracted using program slicing and verified by Murphi tool. Aizatulin *et al.* [14] have proposed model extraction using symbolic execution. These works extract the protocol specifications from the source code, while our approach does not assume to have the source code and provides blackbox fuzzing to infer the semantics when the source code is not available.

Security Analysis on SSO Protocols. Extensive research has been conducted to manually analyze security of SSO protocols. By reverse enginerrring the client implementations, Hanna *et al.* [27] have revealed that some SSO protocols, including Facebook Connect and Google Friend Connect, use the cross-domain communication channel–`postMessage` insecurely, E.Tsyrklevich and V.Tsyrklevich [40] have demonstrated several attacks such as CSRF against the OpenID protocol. Wang *et al.*'s work [42] have conducted a field study on the commercially deployed web SSO systems and discovered 8 serious logic flaws in many notable IDPs and SPs. Xing *et al.* [45] have attempt to protect integrators for their integration of third-party SSO Web services.

Some formal analysis approaches also have been used to analyze the security of SSO protocols. Miculan and Urban [33] manually extract specification of Facebook Connect Protocol from the HTTP messages exchaged. They model the protocol in HLSPL and check it using AVISPA. Bansal *et al.* [17] use applied pi-calculus and ProVerif to analyze the OAuth 2.0 protocol. Their work focuses on constructing concrete attacks from the attack trace reported by ProVerif, and building the operational web attacker model library called WebSpi to map the attack trace to web-site actions. Sun *et al.* [39] also model the web attacker precisely. Sun *et al.*manually extract OpenID 2.0 implementation in HLPSL and verify the model using AVISPA and found CSRF attacks. There are also other formal analysis approaches on SSO protocol. Most of them model the protocol manually based on the protocol documentation or specification, and take into consideration only the network attack model. For example, there have been several formal analysis approaches on SAML SSO protocols [16, 26, 28]. In contrast to these work, AUTHSCAN looks at the security flaws in the implementations.

8 Conclusion

We present AUTHSCAN, an end-to-end platform to automatically recover authentication protocol specifications from their implementations. AUTHSCAN has successfully detected 7 security vulnerabilities in real-world applications automatically. Our techniques assume no knowledge of the protocol specifications being checked and rely on a small set of practical assumptions. We hope further research can lead to tools that recover and check complicated security protocols at the lowest level of their implementation details.

Acknowledgments

We thank our shepherd Venkat Venkatakrishnan and the anonymous reviewers for their insightful comments to improve this manuscript. We also thank Matthew Finifter, Joel Weinberger, Jun Pang, Yacin Nadji, Joseph Hong, Bodhisatta Roy and Mayank Dhiman for their helpful feedback and comments. This research is partially supported by research grant R-252-000-495-133 from Ministry of Education, Singapore, research project "Automatic Checking and Verification of Security Protocol Implementations" and "Research and Development in the Formal Verification of System Design and Implementation".

References

[1] AUTHSCAN. https://sites.google.com/site/ndss2013/.

[2] BrowserID. https://wiki.mozilla.org/Identity/BrowserID.

[3] Facebook Connect Authentication. http://developers.facebook.com/docs/authentication/.

[4] Node.js v0.8.14 Manual & Documentation. http://nodejs.org/api/crypto.html.

[5] What is OpenID. http://openid.net/get-an-openid/what-is-openid/.

[6] Windows Live Messenger Connect, Version 4.1. http://msdn.microsoft.com/en-us/library/ff749458.aspx.

[7] Facebook Connect Used By 250 Million People Per Month. http://allfacebook.com/facebook-connect-used-by-250-million-people-per-month_b25501, Dec. 8, 2010.

[8] Security Vulnerability Allegedly Discovered in Dropbox Client. http://news.softpedia.com/news/Design-Security-Flaw-Allegedly-Discovered-in-Dropbox-Client-194427.shtml, Apr. 11, 2011.

[9] Mozilla jwcrypto. https://github.com/mozilla/jwcrypto, May 13, 2012.

[10] The AVISPA project homepage. http://www.avispa-project.org/, May 13, 2012.

[11] Wolfram alpha. http://www.wolframalpha.com/, May 13, 2012.

[12] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[13] M. Abadi and M. R. Tuttle. A Semantics for A Logic of Authentication (Extended Abstract). In *PODC*, pages 201–216, 1991.

[14] M. Aizatulin, A. D. Gordon, and J. Jürjens. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. In *CCS*, pages 331–340, 2011.

[15] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF*, pages 290–304, 2010.

[16] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *FMSE*, pages 1–10, 2008.

[17] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF*, pages 247–262, 2012.

[18] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW*, pages 82–96, 2001.

[19] B. Blanchet. Computationally Sound Mechanized Proofs of Correspondence Assertions. In *CSF*, pages 97–111, 2007.

[20] B. Blanchet and A. Chaudhuri. Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage. In *S&P*, pages 417–431, 2008.

[21] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions On Computer Systems*, 8:18–36, 1990.

[22] C. J. Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *CAV*, pages 414–418, 2008.

[23] G. Delzanno and P. Ganty. Automatic Verification of Time Sensitive Cryptographic Protocols. In *TACAS*, pages 342–356, 2004.

[24] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[25] D. E.Hammer-Lahav and D.Hardt. The OAuth2.0 Authorization Protocol. 2011. IETF Internet Draft.

[26] T. Gross. Security Analysis of the SAML Single Sign-On Browser/Artifact Profile. In *ACSAC*, pages 298 – 307, 2003.

[27] S. Hanna, E. C. R. Shinz, D. Akhawe, A. Boehmz, P. Saxena, and D. Song. The Emperor's New API: On the (In)Secure Usage of New Client Side Primitives. In *W2SP*, 2010.

[28] S. M. Hansen, J. Skriver, and H. R. Nielson. Using Static Analysis to Validate the SAML Single Sign-On Protocol. In *WITS*, pages 27–40, 2005.

[29] S. Juraj, M. Andreas, S. Jörg, K. Marco, and J. Meiko. On Breaking SAML: Be Whoever You Want to Be. In *USENIX Security*, 2012.

[30] D. Lie, A. Chou, D. Engler, and D. L. Dill. A Simple Method for Extracting Models for Protocol Code. In *ISCA*, pages 192–203, 2001.

[31] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *TACAS*, pages 147–166, 1996.

[32] G. Lowe. A Hierarchy of Authentication Specifications. In *CSFW*, pages 31–43, 1997.

[33] M. Miculan and C. Urban. Formal Analysis of Facebook Connect Single Sign-On Authentication Protocol. In *SOFSEM*, pages 99–116, 2011.

[34] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. pages 141–151. IEEE Computer Society Press, 1997.

[35] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *S&P*, pages 513–528, 2010.

[36] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*, 2010.

[37] D. X. Song. Athena: A New Efficient Automatic Checker for Security Protocol Analysis. In *CSFW*, pages 192–202, 1999.

[38] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.

[39] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically Breaking and Fixing OpenID Security: Formal Analysis, Semi-Automated Empirical Evaluation, and Practical Countermeasures. *Computers & Security*, 31:465–483, 2012.

[40] E. Tsyrklevich and V. Tsyrklevich. Single Sign-On for the Internet: A Security Story. In *BlackHat*, July 2007.

[41] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *WOEC*, volume 2, pages 29–40, 1996.

[42] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *S&P*, pages 365–379, 2012.

[43] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *S&P*, May 2010.

[44] T. Y. C. Woo and S. S. Lam. A Semantic Model for Authentication Protocols. In *S&P*, pages 178–194, 1993.
[45] L. Xing, Y. Chen, X. Wang, and S. Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *NDSS*, 2013.

A Termination of Algorithm 1

We informally argue why the Algorithm 1 terminates. First, since AUTHSCAN uses only one trace t as the basis to generate the $ProSet$ which has a fixed number of local protocols and free variables. The newly generated traces in the fuzzing step do not generate new local protocols and variables, but infer more TML terms over these variables and add new actions. Second, for each HTTP data, AUTHSCAN generates two probes: one in which the data is removed and the other in which the data is mutated. Thus, for a message containing N HTTP data elements, only $2N$ probes are generated. Third, after each iteration (step 4-9), the number of variables inferred is monotonically non-increasing; we can only remove certain variables as redundant data. Finally, by bounding the nesting function depth and number of traces in $trPool$, all searches and fuzzing operates over finite state and must terminate.

B Protocol Extraction

B.1 Extracting BrowserID Protocol

In this section, we detail the process on analyzing `myfavoritebeer.org` to demonstrate how AUTHSCAN extracts model from the implementation. As shown in Figure 5, the traces captured by AUTHSCAN are listed in the first two columns, and the corresponding TML statements inferred are placed in the third column.

From message (2), AUTHSCAN infers the HTTP parameter `csrf` as a nonce. AUTHSCAN also associates user name (`USER`) and password (`PWD`) to represent that they should be matching. From message (4), through white box analysis, AUTHSCAN infers that `spkUser` and $\texttt{spkUser}^{-1}$ are an asymmetric key pair generated by function `generateKeypair()`. In message (5), AUTHSCAN figures out that the HTTP parameter *cert* is encoded as a JSON Web Token (JWT) with each segment separated with "." and encoded with Base64 encoding (as described in Section 4.2). When applying the signature verification algorithm `RSA` over one of the segment (the brute-force search as discussed in Section 4.2), AUTHSCAN finds that it is a signature by *IDP_S* over four data elements occurring previously: $\{USER, spkUser, p, expire\}_{k^{-1}_{IDP\_S}}$. Similarly, in message (6), AUTHSCAN identifies that function `sign()` is used to generate signature $\{j, expire1\}_{spkUser^{-1}}$ and this signature is concatenated with IDP's signature (i.e., *cert*) with function `bundle()`. Afterwards, this concatenation is sent by invoking function `Window.postMessage()`.

B.2 Inferred Protocols

Figure 6 demonstrates the protocols inferred using AUTHSCAN; the inferred models are simplified for readability.

B.3 Precision of Inferred Protocols

We investigate the precision of our inferred protocol, which is possible for two of our case studies, to available documentation and manually-crafted specifications. We find that our protocols are fairly precise, subject to our qualitative analysis.

BrowserID Precision. We compare our inferred specification to the documented description of the protocol online [2]. Our inferred protocol matches closely to the description in the documentation. In some cases, it reveals useful information that is unspecified in the documentation. For instance, the documentation says that, the IDP returns a signed structure containing expiration time in the Step 5 of Figure 6-(a)), but documentation does not precisely specify the duration of the "expiration time". AUTHSCAN finds that the duration is large enough to permit replay attacks that are longer than 726 seconds. This intermediate result is useful for further analysis, such as verification on time sensitive protocols [23].

We find the protocol to match the documentation exactly (subject to our manual interpretation), except for one additional difference. The document states that the SPs are allowed to send the signed data to BrowserID for verification in the specification rather than local verification. Since this message is sent between SP and IDP servers rather than been relayed in the browser, it is not represented in our inferred specification.

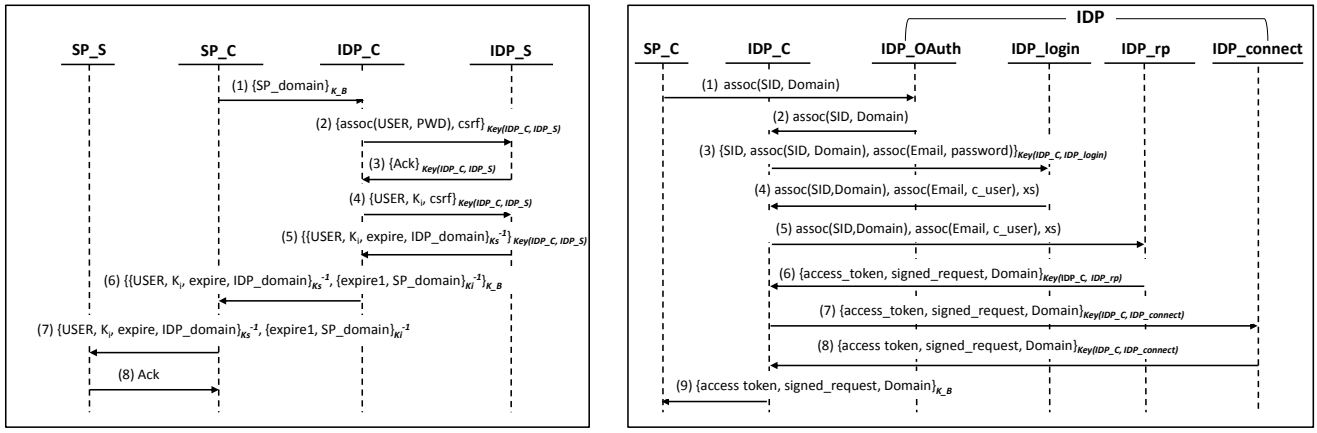
Facebook Connect Precision. Facebook Connect originates from OAuth 2.0 authorization protocol [25]. In Ebay-Classified case, our inferred protocol consists of 11 rounds and 65 parameters (including cookies and GET/POST parameters), comparing to 7 rounds and 11 parameters in the specification. The extra rounds and parameters, which shows our inferred protocol is more precise, may be vulnerable to the protocol and have been analyzed by AUTHSCAN. Furthermore, compared to recent work which manually extracts the Facebook Connect protocol, our model has defined more precisely the terms exchanged in the protocol [33]. Our inferred specification is also more detailed than the prior work of Hanna *et al.* [27]. Finally, we find that our Facebook Connect model is different from the description in Wang *et al.*'s recent work [42]— this is because

#	Input		TML
(2)	HTTP Messages POST https://login.persona.org/wsapi/authenticate_user Host: login.persona.org "email":"alicessotester@gmail.com", "pass":"alice", "csrf":"UaZWfqrQmYwemitM1U8nUw=="	Javascript code snippet NONE	Initial Conditions r has $csrf \wedge p$ has $csrf$ IDP_C(r) NewAssoc({r,p}, $assoc$ ($USER$, PWD)) Send(p, {$assoc$($USER$, PWD), $csrf$ }) IDP_S(p) Receive(r, { $assoc$(M, N), $csrf$ })

(4)	POST https://login.persona.org/wsapi/cert_key
	Host: login.persona.org
	"email":"alicessotester@gmail.com",
	"pubkey":"{\"algorithm\":\"DS\"……6233397a\"}",
	"csrf":"UaZWfqrQmYwemitM1U8nUw=="
		syncEmailKeypair:function(…){…,
		d.withContext(function(){
		a.***generateKeypair***({
		algorithm:"DS",
		keysize:c.KEY_LENGTH}, …)})}
			IDP_C (r)
			NewKeyPair($spkUser$, $spkUser^{-1}$)
			Send(p, $USER$, $spkUser$, $csrf$)
			IDP_S(p)
			Receive(r, M, Y, $csrf$)

(5)	GET https://login.persona.org/wsapi/cert_key Host: login.persona.org "cert":"eyJhbGciOiJSUzI1NiJ9.eyJwdW....SfqAt5…"	NONE	IDP_C(r) Receive(p, X) IDP_S(p) NewNonce($expire$) Send(r, { M, Y, p, $expire$ }$_{k^{-1}_{IDP\_S}}$)
(6)	NONE	assertion.***sign***({},{audience:c,expiresAt:j},g, function(d,g){ k=a.cert.***bundle***([f.cert],g),…}) b.window.***postMessage***(JSON.stringify(a), b.origin)	IDP_C(i) NewNonce($expire1$) Send(j, [X, { j, $expire1$ }$_{spkUser^{-1}}$]) SP_C(j) Receive(i, R)

Figure 5: The HTTP trace of BrowserID and the corresponding TML statements (The full messages are available at [1].)



(a) the Sequence Diagram of BrowerID

(b) the Sequence Diagram of Facebook Connect

Figure 6: The sequence diagrams inferred from implementations of BrowserID and Facebook Connect

their work considers the Flash implementation whereas we analyze the JavaScript-based implementation which works in today's web browsers by default.

C TML to ProVerif Inputs

TML is an high-level abstract model language, which can be directly translated into applied pi-calculus. We do not present the formal semantics translation between these two languages, but intuitively explain the mapping between them. The applied pi-calculus model of the running example (Figure 1 and 3) is shown in Figure 7.

Conversion. Most syntax and semantics can be directly mapped to applied pi-calculus. The initial conditions (initial knowledge of the participants) are represented with a set of global variables (line 17-21), where the terms initially unknown to Z is labeled as `private`, such as `k_IDP_s` (line 18), the private key of *IDP_S*. The cryptographic functions are translated into constructor (`fun`) and destructor (`reduc`) (line 6-15). The local protocols are

represented with the processes (line 33-82), whose identifers are represented with `i,j,r,p` (line 17) of `Host` type (line 1). For the action schema, the Begin* and End* are mapped to `event` (line 67 and 57); the Send and Receive are mapped to `out` and `in`; the *assoc* is represented with the `table` (line 22), and NewAssoc is mapped to `insert` a tuple into the `table` (line 34). However, one problem is that ProVerif does not scale as the number of tables increases. To solve this problem, we also can model the *assoc* using functions. In particular, AUTHSCAN uses the same modeling method as modeling symmetric cryptographic primitives. For example, the $assoc(i, authtoken)$ in Figure 3 is modeled as `mysenc` at line 13-15. Specially, if this *assoc* happens to be a long-lived or guessable token which needs to be added into Z's knowledge set, AUTHSCAN just casts the encryption key to the attacker (`addattackerknow` at line 77-78). The checking action is mapped to the matching action, for example, `let(=M, =N) = checksign(P, spk(k_IDP_s))` (line 42) checks whether `P` is a signature over `(M, N)` using the private key `K_IDP_s`. The channel is slightly different from TML because ProVerif supports both public and private channels. AUTHSCAN translates HTTP into public channel (`ch` at line 23, 38 and 46) which is readable and writable to the attacker; HTTPS and cross-domain communication is translated as private channels (`https` at line 25 and 48, and `browser` at line 24 and 40).

For the syntax or semantics not supported by ProVerif, AUTHSCAN models them in alternative ways. For example, ProVerif does not support a writable but non-readable (for the attacker) or a readable but non-writable channel. When AUTHSCAN finds that the sender `origin` of `postMessage` is not checked (such as Step ② in Figure 1), which means this channel becomes an attacker-writable channel (but remains unreadable), it turns the `browser` channel writable by adding an input before `out` messages to `browser`, as shown at line 38-40. Conversely, if it finds that the channel is readable, it adds an `out` after in message from the channel. Finally, after we fixing all the vulnerabilities, ProVerif reports that the protocol is verified.

Detected vulnerabilities. ProVerif detects three attacks in this model. First, it reports that the attacker can derive the `token` using the key `k_i_j_com` cast to his knowledge set (line 77-78). After "fixing" this flaw (Here fixing means correcting the flaw in the model instead of in the implementation) as shown at line 74-78, it reports a replay attack where the attacker can obtain the `token` from line 46, and then replay it to line 54. After "fixing" this flaw using HTTPS to replace HTTP as shown at line 48 and 55, ProVerif reports the MITM attack shown in Section 2.1. The attacker replaces `mynext` at line 38 and finally gets the `token` from line 63.

```
1  type Host.
2  type key.  (*symentric key*)
3  type spkey.(*public key*)
4  type sskey.(*pivate key*)
5
6  (* Shared key encryption *)
7  fun senc(bitstring, key):bitstring.
8  reduc forall x:bitstring,y:key;sdec(senc(x,
       y),y)=x.
9  (* Signatures *)
10 fun spk(sskey):spkey.
11 fun sign(bitstring, sskey):bitstring.
12 reduc forall x:bitstring,y:sskey; checksign
       (sign(x,y), spk(y)) = x.
13 (*fun*)
14 fun mysenc(Host, key):bitstring.
15 reduc forall x:Host,y:key;mysdec(mysenc(x,y
       ),y) = x.
16
17 free i, j, r, p:Host.
18 free k_IDP_s:sskey [private].
19 free k_i_j_com:key [private].
20 free sp:bitstring.
21 free sessionID, CSRFToken:bitstring[private
       ].
22 table sp_table(Host, bitstring).
23 channel ch.
24 free browser:channel [private].
25 free https:channel [private].
26
27 event BeginInit(Host).
28 event EndResponse(Host).
29
30 query x:Host, y:Host; inj-event(EndResponse
       (x)) ==> inj-event(BeginInit(y)).
31 query attacker(mysenc(i, k_i_j_com)).
32
33 let SP_C = (*i*)
34  insert sp_table(j, sp);
35  (*******************************
36       3. Fix postmessage flaw
37   *******************************)
38  (*in(ch,(j:Host,sp:bitstring,mynext:
       channel)); *)
39  new mynext:channel;
40  out(browser,((j,sp),mynext));(*Step 1*)
41  in(mynext,(M:Host,N:bitstring,P:bitstring)
       ); (*Step 4*)
42  let(=M, =N) = checksign(P, spk(k_IDP_s))
       in
43  (*******************************
44       2. Fix HTTP replay attack
45   *******************************)
46  (*out(ch, (M,N))*)
47  in(ch, (M:bitstring, N:bitstring));
48  out(https, (M,N))(*step 5*).
49
50 let SP_S = (*j*)
51  (*******************************
52       2. Fix HTTP replay attack
53   *******************************)
54  (*in(ch,(M:Host,token:bitstring))*)
55  in(https,(M:Host,token:bitstring));(*step5
```

```
      *)
  let (=M) = mysdec(token, k_i_j_com) in
  event EndResponse(i).

let IDP_C = (*r*)
 in(browser,(X:bitstring,Y:channel));(*step
      1*)
 out(https,(X,sessionID,CSRFToken));(*step2
      *)
 in(https,(M:Host,N:bitstring,P:bitstring))
     ;(*step 3*)
 out(Y, (M,N,P)). (*step 4*)

let IDP_S = (*p*)
 in(https, (X:bitstring, =sessionID, =
     CSRFToken));   (*step 2*)
 event BeginInit(j);
 let(M:Host, Mdomain:bitstring) = X in
 get sp_table(=M, =Mdomain) in
 let token = mysenc(i, k_i_j_com) in
 let idpsign = sign((i, token), k_IDP_s) in
 out(https, (i, token, idpsign)).(*step 3*)

 (*****************************
       1. Fix guessable token
  *****************************)
let addattackerknow =
  (*out(ch, k_i_j_com)*)
  new padding:bitstring.

process
 (!SP_C|!SP_S|!IDP_C|!IDP_S|!
     addattackerknow)
```

Figure 7: Applied pi-calculus model of the running example

D CSRF Attack Script

The following script can be used by the attacker to commit a CSRF attack, which modifies the content on the web pages of `Myfavoritebeer` without the user's approval.

```
<iframe name="formFrame"></iframe>
<script>
 formFrame.document.body.innerHTML=
  '<form name="tfm" action="http://myfavorite
    beer.org/api/set" method="post" target=
    "_parent"> <input type="text" name="beer"
    value="Hello Kitty"/><input type="submit
         "/></form>';
 formFrame.document.all.tfm.submit();
</script>
```