InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations

Luyi Xing, Yangyi Chen, XiaoFeng Wang
Indiana University Bloomington
Bloomington, IN, USA
{*luyixing, yangchen, xw7*}*@indiana.edu*

Shuo Chen
Microsoft Research
Redmond, WA, USA
shuochen@microsoft.com

***Abstract*—A web application today often utilizes web APIs to incorporate third-party services into its functionality. Such API integration, however, is full of security perils: recent studies show that popular web sites using high-profile web services, such as PayPal/Amazon checkouts and Facebook/Google single-sign-on (SSO) services, are riddled with logic flaws, enabling a malicious party to shop for free or log into a victim's account. To address this new threat, techniques need to be developed to facilitate secure integration of third-party web services.**

To answer this urgent call, we present in this paper *InteGuard*, the first system that offers security protection to vulnerable web API integrations. InteGuard operates a proxy in front of the service integrator's web site, performing security checks on a set of invariant relations among the HTTP messages the integrator receives during a transaction (e.g., a checkout from a web store or a web SSO). These invariants link multiple HTTP sessions to a transaction and capture their security-critical relations. They also characterize transaction-related communication the proxy cannot directly observe, which happens between the client and the service provider. InteGuard includes a suite of novel techniques that automatically extract such invariants from a variety of communication channels adopted by diverse integrations and achieve effective false positive control in this process. Our evaluation shows that InteGuard can defeat complicated exploits on high-profile web services, with little impact on their normal operations.

I. Introduction

Recent years have witnessed the growing trend of moving computing from the desktop to the web, which gives rise to a huge demand for web applications. To meet such a demand, web developers increasingly utilize existing web services (e.g., search, map, payment, authentication, etc.) to rapidly build up their own applications, through integrating the Application Programming Interfaces (APIs) provided by these services. For example, many web stores (e.g., `Buy.com`) call third-party payment APIs provided by cashier services such as PayPal, Amazon Payments and Google Checkout; more and more websites include single sign-on (SSO) APIs of Facebook, Google, Yahoo, Twitter, PayPal and others to let their customers log in through these identity providers (IdP). As a result, today's web applications become increasingly hybrid, with their program logic distributed across multiple parties, including not only the websites hosting these applications and their clients but also various third-party API service providers. Such a development, however, has made the web applications complicated and error-prone, as discovered in our recent studies [1], [2].

Logic flaws in web-API integrations. The unique security challenge such a *hybrid web application* (i.e., those integrating third-party web services) faces is how to securely coordinate different parties it involves, including the one that integrates the API services (called *integrator*), the provider of the services and the web clients using the application. Our prior research shows that this is very difficult to be done securely: given the diversity in the ways that an API service is integrated and the partial view the integrator and the API provider share about each other's code and runtime state, logic flaws in API integration are hard to avoid and can often be easily exploited by a malicious party to cause miscoordinations and bypass security protection [1]. An example is NopCommerce's integration of Amazon Payments, in which a shopper is supposed to place an order on the web store powered by NopCommerce and pay for the order on Amazon. What has been found is that miscoordination can be introduced by a malicious shopper who exploits a logic loophole in the integration to convince the store that the order has been paid through Amazon and Amazon that the payment should be made to the shopper's own Amazon seller account [1]. As another example, some websites' integrations of PayPal Access SSO are found to be evadable, simply by convincing the integrators that a message field signed by PayPal is a user's PayPal ID and PayPal that it should contain the person's mailing address [2].

The studies we conducted in the past years show that such logic flaws are pervasive [1]: leading web stores (`Buy.com`, `JR.com`), popular merchant systems (NopCommerce, Interspire) and even mainstream payment service providers (Amazon) all contain loopholes in their integration code that enable miscreants to shop for free; important websites integrating SSO mechanisms of Facebook Connect, Google ID, PayPal Access, etc. are vulnerable to the threat that allows unauthorized parties to log into victims' accounts [2]. The problem cannot be solved by solely relying on existing techniques like protocol verifications, due to the diversity in the ways an API service is incorporated and used by different integrators, and the limited understanding each party (the integrator, the provider) has about the other's

behavior (e.g., how the other party verifies a signed message) and the underlying runtime system (e.g., whether Adobe Flash allows cross-domain communication [3]). So far, little has been done to address this emerging security challenge.

Integration protection. Although the above security flaws can be introduced by all parties involved in an API integration, prior research [1], [2] shows that the weakest link remains on the integrator side, whose code appears to be way more error-prone than that of the provider. Effective integrator protection, therefore, will significantly mitigate the threats posed to vulnerable integrations. In our research, we made the first attempt to find such a solution, with an aim to develop a generic and automated technique to protect integrations from exploits. Note that our approach cannot count on the availability of integration-related source code, as the integrator does not have provider-side code and often utilizes off-the-shelf integration solutions such as software development kit (SDK) released by the provider or commercial software (e.g., Interspire). The approach is also expected to work independently of integrations' semantics (e.g., payment, SSO), which needs human effort to specify.

An observation we can leverage is that for a specific integrator-side implementation, its interactions with the service provider are rather mechanic: given a small set of client inputs (e.g., items to be purchased, usernames, etc.), the HTTP messages to be exchanged during a multi-party integration-related operation like checkouts or SSO (called a *transaction* in the paper) are predictable and the parties involved typically do not perform complicated transformations on the content of the messages they receive for response generation. This makes us believe that the invariant relations among those messages' parameters can be identified and utilized to avoid the situation that necessary security checks fall through the cracks.

Program invariants have been used to detect faulty application logics. For example, Waler [4] infers likely invariants from a web application's normal operation and analyzes its source code to find violations. The approach is language-specific and needs source code, which is often unavailable to an integrator-side protection, particularly when it comes to the provider's program. Simple traffic invariants (e.g., session parameters) are employed by BLOCK [5] to detect the exploits on the websites without proper access-control protection. Different from such prior work on two-party web applications, the problem we face, which involves three parties (the integrator, the provider and the client), is way more complicated, requiring identification and utilization of new types of invariants that cannot be handled by those approaches. As an example, consider a checkout transaction, which involves multiple HTTP sessions between the merchant (the integrator), the shopper (the client) and PayPal (the provider). Invariant relations among the content exchanged in different sessions (e.g., order ID) need to be found so as to link them to the checkout and ensure their consistency (e.g., payment matching price) in the presence of other checkouts and unrelated activities (e.g., choosing items, etc.). Even more challenging, the merchant-side protection cannot see the session between the shopper and PayPal (unlike Waler, which sees the whole program, and BLOCK, which sees all the traffic), and has to infer its content from what the merchant sends to the shopper. This requires extracting the parameters related to the shopper-PayPal interaction from the merchant-shopper communication channels, which are very diverse in different integrations of even the same API (e.g., HTTP redirect, form, JavaScript, JSON, etc.). Also, the efficacy of this protection critically depends on effective false positive control of traffic invariants, which needs a novel solution.

Our work. In this paper, we present a suite of new techniques to address these challenges, making a first step toward automatic protection of vulnerable web API integrations. Our approach, called *InteGuard*, performs security checks on transaction traffic forwarded by a proxy sitting in front of the integrator's website. Such a proxy can simply be a load balancer operated by most large websites to distribute web traffic to their clusters. For smaller websites, it can be run by a trusted intermediary, such as Janrain that integrates major SSO services to offer its customers convenient access to them [6]. Attached to the proxy is our InteGuard server, which inspects invariants within web traffic before it reaches web servers. The invariants we look at are specified on the parameters automatically identified through a simple data-flow analysis that links what the browser can see to the communication channels the InteGuard server observes, and extracted from such channels through an in-depth analysis on related HTTP messages, HTML/XML files, JavaScript code, etc. From these parameters, invariants are automatically generated to bind transactions to an integrator, messages to a transaction and each message within a transaction to its legitimate successor. InteGuard achieves effective false-positive control through strategic selection of normal traffic traces for invariant generation and carefully-designed differential analyses on the traces.

We put our design to the test against 11 real-world exploits reported by prior research [1], [2]. All these exploits work on subtle logic flaws in popular hybrid web applications, including leading merchant software integrated with PayPal, Amazon Payment and Google Checkout, and high-profile SSO systems supported by Facebook, Google, PayPal, etc. InteGuard defeated such complicated exploits (see [1], [2]) and worked effectively on different types of hybrid web application (web stores, SSO) at minimum human intervention, even though our design has not been tied to the specific features of these applications and rather the way how web API integrations work (see Section II-A). When evaluating our approach on 1000 normal transactions with

diverse content through those web applications, we did not observe any false positives. Also, a performance test on a prototype we built shows that InteGuard has only a small impact on the operations of the websites it protects.

Contributions. The paper's contributions are as follows:

• *Toward automated and generic integration protection*. InteGuard is the first approach that offers practical supports to secure integration of multiple web services. It made a first step towards automatically protecting different types of integrations (e.g., payment, SSO). This is achieved through a suite of novel techniques that comprehensively analyze the communication observable to the integrator during a transaction, identify the invariants characterizing the whole transaction and effectively control false positives.

• *Implementation and evaluation*. We implemented InteGuard and evaluated it with real exploits, normal transaction traffic and a performance test, which demonstrates that our approach indeed offers practical protection.

II. Background

A. Web API Integration: How It Works

To incorporate a third-party web service, a hybrid web application needs to invoke the API functions of the service with a set of parameters. These parameters include the description about how the service should be provided and how its outcomes can be sent back to the integrator. For example, an integration of Amazon Simple Pay needs to call the API https://amazon.com/paypipeline, using parameters such as `AccountId`, `amount`, `OrderID`, `returnURL` (a link to an integrator-side function for receiving the return of the service) and others. This process involves the client of the web application (Figure 1).

Communication in a transaction. During a transaction, the client communicates with both the integrator and the service provider using HTTP request/response pairs (typically through the HTTPS channels the client establishes with these parties), which we call *HTTP round trip* or simply *RT*. The servers (the integrator or the provider) can also directly interact with each other through RTs. In Figure 1, RT1, RT2 and RT3 are client-server RTs and RT4 happens between the integrator and the provider. Such multi-party HTTP based communication can become complicated. Specifically, an integrator can hold an HTTP request from the client and generate a set of HTTP RTs to query the provider for the client's information before sending back the response. This happens, for example, when a web store asks for a shopper's payment information from PayPal. Moreover, a server often utilizes an HTTP response to the client to trigger a browser-side request, which invokes a web API on the same or a different server. An example is the aforementioned integration of Amazon Simple Pay, which allows a web store to send a shopper a 302 redirect to call the Amazon API.

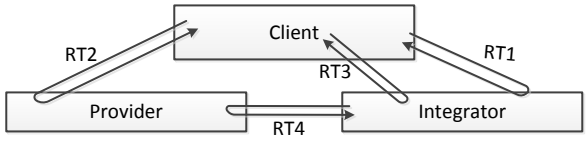

Figure 1. A three party communication example described by HTTP round trips (RTs).

Such server-server indirect communication, i.e., an HTTP response that causes a browser to generate an HTTP request, is called *browser-relayed message* (BRM) in the prior work [2]. It has been extensively used in web API integrations to bind the client to a transaction, particularly in the cases of online shopping checkouts ("who should pay for the order") and SSO ("who holds the ID the IdP vouches for"). This concept can help simplify the description of complicated transaction communication: for example, Figure 2 illustrates a BRM view of the communication in Figure 1; here BRM1 (RT1 response $\rightarrow$ RT2 request) and BRM2 (RT2 response $\rightarrow$ RT3 request) describe the most important part of the communication through RTs. In the rest of this paper, we use BRMs or RTs under different circumstances, for the convenience of presentation.

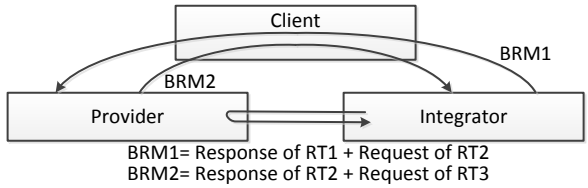

Figure 2. The same communication example with client-initiated RTs simplified as BRMs.

Diversity in integration. Web APIs do not come with a set of restrictions on how they should be integrated. The ways they are utilized in web applications are actually quite diverse. Particularly, the indirect communication between the integrator and the provider can be performed differently in different systems: a BRM can be as simple as a 302 redirect or a meta refresh redirect to conduct an HTTP `GET` to the target server, or happen through more complicated channels such as an HTML form delivered to the client's browser to produce an HTTP `POST` to the server, a script sent to the browser to generate the form or even a JSON message acquired by the script from one server for building an HTTP request to the other server. Moreover, it is completely up to the integrator how to use the outputs of these APIs. For example, what comes out of a payment API is just the information about whether an order is paid or not. How to act on it (e.g., updating the status of the order, mapping the order to the items purchased, etc.) is decided by the integrator's system. Prior research [1] shows that this part of integration could introduce serious security problems if

it is not well thought out.

B. Integration Logic Flaws and Exploits

Cashier as a Service (CaaS). Prior research found that online merchants' integrations of checkout services, called *Cashier-as-a-Service* or CaaS, are riddled with logic loopholes, which often allows a malicious shopper to shop for free [1]. As an example, consider the integration of Amazon Simple Pay within NopCommerce, a leading merchant application [7]. Figure 3 (BRM view of transaction traffic) and Figure 4 (RT view) illustrate how the integrator (merchant) works with the client (shopper) and the service provider (CaaS) through two BRMs (essentially two browser redirections) after the client places an order to the merchant (referred to as jeff.com) through an HTTP request (RT1.request in Figure 4). This request triggers BRM1 (a redirection, RT1.response to RT2.request, from the merchant to Amazon through the client), which the merchant uses to call the API https://amazon.com/paypipeline with signed order information, including `orderID` and the gross amount, etc. BRM1 also specifies a merchant side API through its `returnURL`: jeff.com/finishOrder, which Amazon calls after the payment is done. As the result of BRM1, the client's browser is taken to Amazon's website, where the shopper pays according to the order information. Since BRM1 is signed by Jeff, Amazon transfers the payment from the shopper's account to Jeff's account. Then, Amazon invokes jeff.com/finishOrder according to `returnURL`, through a signed BRM2 (RT2.response to RT3.request) that brings the browser back to the merchant's website and also notifies Jeff of the completion of the payment. The merchant then verifies Amazon's signature and sets the order status to "PAID".

This integration turns out to be vulnerable. Suppose that a malicious shopper also has a seller account with Amazon under the name "Mark". Then what he can do is to pay Mark (actually, himself) but check out an order from the merchant Jeff (https://jeff.com). Specifically, when BRM1 goes through the client, the client can simply remove Jeff's signature and use Mark's identity to sign the message. As a result, from Amazon's standpoint, the transaction becomes a purchase the shopper makes from Mark, so the payment is made to Mark. However, after the payment is complete, Amazon still redirects the browser to send its signed notification to Jeff according to `returnURL` on the BRM1 it receives. Although this notification is about the shopper's payment to Mark, the program logic of `finishOrder` (on jeff.com) only checks `orderID`, which is identical to Jeff's pending order. This will cheat the merchant into believing that his order has been paid. The prior study shows that the only way for Jeff to realize the problem is to check `payeeEmail`, an email address associated with his Amazon seller account. However, Amazon never states that this address should be verified in a transaction.

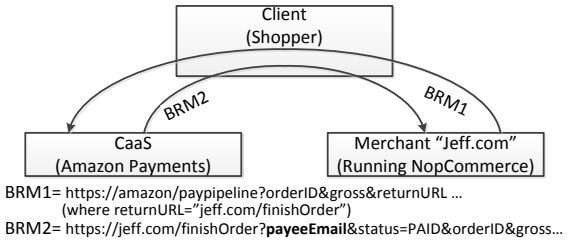

Figure 3. BRM based view of NopCommerce's integration of Amazon Simple Pay.

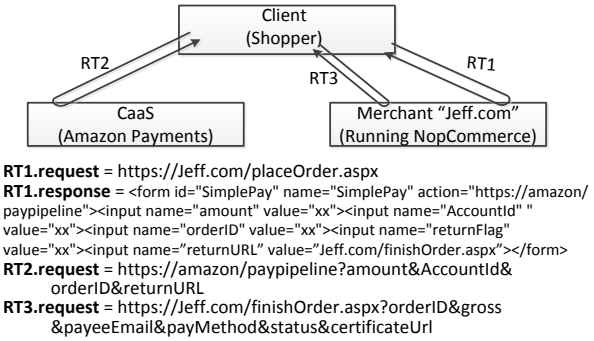

Figure 4. Round-Trip view of the integration of Amazon Simple Pay.

Web SSO. A recent study reveals critical logic flaws in web SSO systems that integrate the APIs of Facebook, Google, PayPal and others [2]. An example is illustrated in Figure 5, which describes the SSO service Google provides to Smartsheet (smartsheet.com) through Google ID, which is based on the OpenID standard [8]. During an SSO transaction, a client who wants to log into smartsh eet.com is redirected to Google through BRM1, which invokes an API with a set of parameters. Among them, `openid.ext1.required` is a list that contains a set of identity information of the user delegated by the client Smartsheet asks Google to vouch for. The list includes the user's first, last names and most importantly her email address. After the user logs in her Google account, Google uses BRM2 to reply to Smartsheet with signed parameters, as recorded in its `openid.signed` list. Smartsheet verifies Google's signatures on these parameters before signing the client in.

A logic flaw in this SSO integration is that the integrator does not verify the consistency between `openid.ext1.required` (BRM1), i.e., the list of items it asks Google to verify, and `openid.signed` (BRM2), i.e., what Google actually signs in its response. Therefore, a malicious client can change the content of `openid.ext1.required` to drop the email in BRM1, which causes Google to only sign on the user's name, and then append the attacker's (unsigned) email to BRM2. When this happens, the integrator only verifies the signature on the user's first and last names. Once succeeded, it still treats the email added by the attacker as the user's identity. This

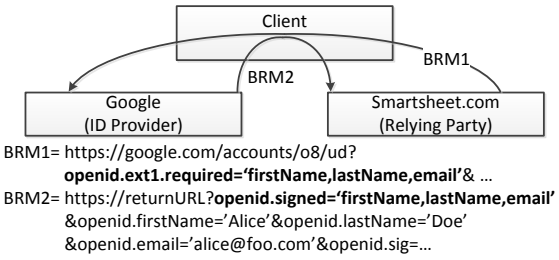

Figure 5. Integration of Google SSO.

enables the attacker to log into any legitimate user's account on the integrator's website. This flaw actually exists in many websites and both Google and OpenID foundation published security advisories on the problem [9], [10].

Another interesting example is OpenID's type confusion problem. Consider the integration of PayPal Access in Shopgecko.com as an example [2]. It works just like what is illustrated by Figure 5, except that now Shopgecko is the integrator and PayPal is the provider. BRM1 shown in the figure carries `openid.ext1 .type.email` (`type.email` for short), which is actually an OpenID type [11] that determines the value of `openid.ext1 .value.email` (`value.email`) in BRM2: if the content of `type.email` in BRM1 is "PayPal ID", PayPal will fill in `value.email` (in BRM2) with a user's PayPal ID and sign the field; if `type.email` is set to other types, for example, mailing address, the user's address will be entered into `value.email` by PayPal. Shopgecko always uses PayPal ID in its `type.email`. However, a malicious client can change it to the type of mailing address through BRM1 and log in her PayPal account. As a result, PayPal fills the adversary's address (which can be maliciously set to a victim's PayPal ID) to `value.email` in BRM2, while the integrator still treats the content of `value.email` as the victim's PayPal ID. This allows the attacker to log into Shopgecko.com as the victim.

C. Adversary Model

The techniques presented in the paper aim at mitigating the threat to integration related logic flaws, rather than more generic flaws within web applications. We focus on the integrator's front, as its program logic was found to be the main source of such security flaws [1], [2]. Our study assume that the service provider is trusted, though it can be misled by the adversary to behave unexpectedly. We do not trust the client: it can be fully controlled by the adversary or under the influence of the malicious dynamic content (e.g., JavaScript) running in its browser.

III. Design and Implementation

A. Overview

Architecture. The architecture of InteGuard is illustrated in Figure 6. It includes three components, a trace collector (Section III-B), a security policy generator (Section III-C) and an ICAP (Internet Content Adaptation Protocol) server attached to a standard reverse proxy (Section III-D). The trace collector is used by a human user to perform a few well-designed tests (4 to 5 transactions), which produce a set of labeled traces (including the HTTP messages of these transactions) for invariant identification. This testing process is designed to generate high-quality traces that help minimize false positives. The collector was implemented in our research as a Firefox add-on and a component running in the ICAP server to intercept the direct communication between the integrator and the provider. The policy generator works on the traces collected to automatically extract invariants and construct a finite state machine (FSM) that represents the security policies for the integration. The ICAP server inspects HTTP messages received by the integrator's website and enforces the security policies. We utilized *Squid* [12] and *GreasySpoon* [13] as our proxy and ICAP servers respectively, and specified FSM in GreasySpoon script.

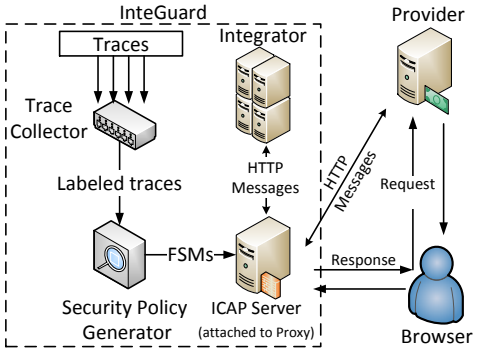

Figure 6. Architecture of InteGuard.

An example. Consider NopCommerce's integration of Amazon Simple Pay in Figure 3 and Figure 4. Throughout the rest of the paper, we use this integration as a running example. To produce its traffic traces, one needs to do four different purchases through the collector to ensure the diversity in the contents of the traces collected. More specifically, these traces are generated using different inputs the client can set through its merchant and Amazon user interfaces, such as product name, quantity, etc., and also cover different integrators, different transactions, different users, etc. From these traces, the collector then automatically extracts various types of content parameters, such as the API name *placeOrder*, `orderID`, `gross`, `payeeEmail`, etc., which we call *elements*. Most important here is to link the parameters the browser uses to call the Amazon API in RT2.request (i.e., the client-provider message in Figure 4), which InteGuard cannot directly observe, to the content of RT1.response (i.e., the proxy-client message in the figure), which our ICAP server can see. This is achieved through a simple data-flow analysis that bridges the content of these

two HTTP messages (see Section III-B). This step involves automatic parsing and analysis of different channels such communication goes through, simple redirects or complicated script-based BRMs different integrations adopt. Such an analysis converts individual traces into a set of vectors, each of which includes the elements within an HTTP RT.

Over those vectors, the security policy generator performs a series of differential analyses to identify different types of invariants, including *integrator-specific invariants* such as `payeeEmail`, *transaction-specific invariants* such as `orderID` that connects multiple messages together and *local invariants* such as `gross` (messages involved in BRM1 and BRM2 in this case). These invariants ensure that an inbound message the integrator acts on belongs to the right transaction performed by a right integrator, and is exactly the one expected by the current transaction state. The analysis also fingerprints the beginning and the end of a transaction. All such invariants are used to build an FSM that the ICAP server runs to perform security checks on the traffic it receives. In the rest of the section, we elaborate on these steps and their related components.

B. Trace Collection and Analysis

Even before invariants can be identified, we need to come up with nontrivial solutions to a couple of technical challenges. First, collection of the HTTP traces of normal transactions is more complicated than it appears to be: consider that we randomly pick up two checkout transactions involving the same user; then user-specific information may end up in the invariant set identified, a false positive we want to avoid. Second, given that InteGuard only works on integrator-side traffic and does not know how the integration works *a priori*, it becomes challenging to extract the parameters for an API to be invoked on the provider side from the integrator's response to the client (e.g., RT1.response in Figure 4), which is part of the BRM (e.g., BRM1 in Figure 3) triggering that API, simply because the response can take various forms and these parameters (i.e., elements) could be hidden deeply within an HTML form, XML data, JSON parameters or even JavaScript code. Here we explicate how our trace collector deals with these issues.

Trace collection. To avoid the false positive mentioned above, we need the content of the HTTP traces used for invariant detection to be as diverse as possible. To this end, InteGuard requires four test transactions to be executed through its collector, i.e., the ICAP server with a trace-collection module and a browser with a trace-analysis add-on. Such a small set of traces are found in our research to be sufficient for capturing security-related traffic invariants in most integrations because in most cases, the communication among the client/integrator/service provider during a transaction is very mechanic, which does not involve complicated human interactions and often follows a fixed HTTP traffic sequence like what happens in a network protocol. There are exceptions, though: occasionally, an integration may include human interactions in the middle of a transaction. For example, a variation of PayPal Express lets the user first register a transaction token from PayPal and then go back to the web store's site to enter shipping and billing addresses before she can finalize the transaction. To avoid the noise introduced by such interactions to a transaction, our current treatment is asking the party who produces those test traces to label human interactions (through our browser add-on) so as to avoid generating some invariants (actually, the call-sequence invariant, see Section III-C) that could cause false positive.

Note that this trace collection process is the only step in our system that needs human intervention: a human user can use the browser to perform these transactions, e.g., checking out items from a web store through PayPal, and let the collector record the traces. It is possible to automate the step using web-testing tools such as Apache JMeter [14] once transaction parameters are set, which we will study in the follow-up research.

Specifically, before the collection, the integrator is supposed to open two accounts with the service provider[1], which gives it two different identities. Under each of them, a group of two transactions are carried out. In one transaction, we avoid the same content for two "attributes of interest" when possible: for example, the shipping and billing addresses need to be different. Within one group (associated with the same account), the values of such attributes on one transaction are set differently from those of the other transaction. These attributes are the inputs to the transactions and can be acquired from the user interfaces of the integrator and the provider websites. More specifically, for a CaaS integration, a user is supposed to give her username for login, choose items from the integrator and go to the provider to pay. Therefore, the parameters she set during this process, such as her account information and the name/price/quantity of the commodity, can be such attributes of interest. For an SSO integration, on the list is just the username of the individual who wants to log in. Note that a list of such attributes is the only semantic information InteGuard needs.

For two corresponding transactions in different groups, we take the same values for their attributes of interest. Specifically, Transaction 1 in Group 1 shares the same parameters with Transaction 1 in Group 2, and the same happens to Transaction 2 in each group. This ensures that the only difference between the corresponding transactions in the two groups is the integrator's identity. Figure 7 illustrates the subset of the four traces collected for analyzing our running example in Figure 4. These traces can be leveraged by a suite of differential analyses for false positive control

[1]This can be done within the development environments some providers offer. An example is PayPal's Sandbox [15], which allows one to have many free accounts.

(Section III-C). The complete traces for the integration and some other examples are available at [16].

Figure 7. Part of the traces for the running example. T_1 and T_2 are in Group 1 (Integrator 1), and T_1' and T_2' in Group 2 (Integrator 2). These traces are simplified. The complete ones are here [16]

When a client wants to stop a transaction in the middle, for most integrations, it just needs to browse away from transaction-related pages or simply close related windows/tabs. On the other hand, some integrations do provide links that let the client terminate ongoing transactions. An example is NopCommerce's integration with PayPal Payments Standard. In this case, we collect an additional "abort" trace for a transaction that utilizes the same set of parameters as Transaction 1 in Group 1, except that it is terminated by the client on the provider side (the only place throughout a transaction where the client's inputs are expected) before the transaction runs to completion.

Element extraction. When processing a test transaction, the collector also performs a preliminary analysis on HTTP messages[2] to extract the parameters of web API calls and key elements from the responses of the calls. This operation converts a raw HTTP trace to the form that the policy generator can work on. Specifically, a trace T the collector outputs comprises a sequence of vector $V_{j=1,\cdots}$. Each vector describes an HTTP round-trip (i.e., its request/response pair): $V_j = (f, m, d, e_1, \cdots, e_n)$, where f is the API function (if any) invoked, m is the HTTP method (e.g., `GET`, `POST`) used, d is the initiator-server pair for this RT, and $e_{k=1,\cdots,n}$ is an element extracted from the RT. Such an element is represented by a triplet (t, p, v), with t specifying the type of the channel that transports the element (e.g., HTTP redirect, HTML form, JavaScript, etc.), p describing the position of the element in a message and v denoting the value of the element. For example, consider an integration that utilizes an HTTP 302 redirect to Amazon.com/paypipeline?orderID= 123&amount= `12&`$\cdots$. The elements here include (302, `orderID`, 123) and (302, `amount`, 12), as the names of the elements (`orderID`, `amount`) can be used to directly locate them in a 302 redirect.

The elements of interest often appear on the request part of an HTTP RT, as parameters for a web API call. These elements are well formulated in the standard URL format[3] (see the above example), and can be easily extracted. This also happens to the HTTP 302 redirects (within HTTP responses) used for the indirect communication between the integrator and the server (e.g., the BRM above). However, other BRM channels are more complicated: particularly, the response from the integrator to the client can include meta refresh redirect, HTML form (see Figure 4 RT1.response), JavaScript and JSON etc., as long as such content generates an HTTP request to call the provider-side API. Even though the request's parameters can be easily identified, the ICAP server cannot see it directly and instead needs to work on the diverse content within the response (from the integrator) that generates the request. Therefore, the challenge becomes how to automatically locate elements in such content.

Our solution is to perform a simple data-flow analysis on a BRM within the browser, which links the content of an HTTP response (e.g., RT1.response in Figure 4) to that of the HTTP request (e.g., RT2.request) the response causes the browser to produce. Our browser add-on first extracts the API parameters from the request and then tries to find the sources of these parameters in the response. This requires parsing the response content into a structure like HTML DOM tree or Abstract Syntax Tree (AST) for JavaScript, and matching the parameters to the values of the elements within the structure. Once succeeded, the collector records the type of the structure and element positions, which are used by the ICAP server during its runtime to locate these elements from the integrator's response for security checks. Here we elaborate how our collector identifies and processes different types of BRM channels.

• *Simple URL redirection*. Standard HTTP 3xx redirection can be directly identified from the header of an HTTP response. The elements involved are within the URL in the `Location` field, and can be extracted according to its standard format. Redirections can also go through HTTP refresh, which can be determined from the HTTP refresh tag the response carries. The target URL here can be found from the `Refresh` field. A little more complicated is

[2]Note that different from prior research on reverse engineering of malware protocols [17], InteGuard is meant to work on the HTTP protocol and typical web content (e.g., HTML) utilized by legitimate web services, whose formats are quite standard.

[3]For a POST request, the URL can be included in its content with its `Content-Type` set to x-www-form-urlencoded. Occasionally, other formats can be used, which are also well defined through `Content-Type`.

meta refresh, whose tag (`meta http-equiv`) and redirect target URL are embedded in the HTML header of the response content. To locate the URL, we can parse the HTML header. A more efficient alternative our collector uses is simply searching the content for the string that contains the HTTP path/application name of the URL used in the HTTP request to the provider, which is observed by our add-on. The string is then parsed to extract elements.

• *HTML form*. HTML form is another common channel for transporting BRMs. Specifically, the integrator includes a form in its response to the client. The client's browser parses the form, which generates an HTTP request (typically a `POST`) to invoke a provider API. To work on this channel, our collector parses the HTML content carried by a response into a DOM tree (using Mozilla's `HTMLParser`) to inspect its form object. What it does is to extract individual form elements (using `getElementsByTagName`) and save the DOM path (form attributes $\rightarrow$ input tag attributes) of the tags whose values match the parameters in the request part of the BRM. This path, which includes form attributes (`id`, `name`, `action`) and the input tag's `name` attribute, are used by a Java HTML parser to locate the elements within the ICAP server during its operation.

An example in Figure 8 shows that the HTML form sent to the client via RT1.response in Figure 4 is converted into elements using the API parameters of RT2.request.

• *JavaScript and JSON*. Even more complicated is the JavaScript-based BRM channel: the integrator dispatches a script to the client's browser, which creates a `POST` to invoke the server's API. The challenge is that the elements of interest here are hidden inside the JavaScript code. Theoretically, identifying the sources of the response's URL parameters requires a thorough analysis on the script's information flows, which is hard. Practically, however, these scripts are typically simple and carry these elements in their constants. This allows us to locate them by parsing the script (within the `<script>` and `</script>` tags on an HTML page) into an AST (using the SpiderMonkey engine) to inspect its constants and record the program positions of those matching the parameters. Note that we do not even need to fully parse the script code, particularly the library functions it calls, as all we need here is just finding out those constants, which are almost always within the script tags on an HTML page.

Another BRM channel is JSON. A script running in the client's browser can get a JSON message from the integrator (`Content-Type`: application/json) and build a response from the message. Our browser add-on runs a JSON parser [18] to process the content of such a response and uses field names to locate relevant elements.

• *Other channels*. XML can also be used in BRMs or Round Trips generated by direct communication between the integrator and the service provider to transfer elements of interests. Our collector can identify the channel from the HTTP Header `Content-Type` and then parse the content to identify these elements. For example, Interspire's integration of Google Checkout uses an XML-based redirection as its BRM. Moreover, the message directly exchanged between a web store using this integration and Google is also in an XML format.

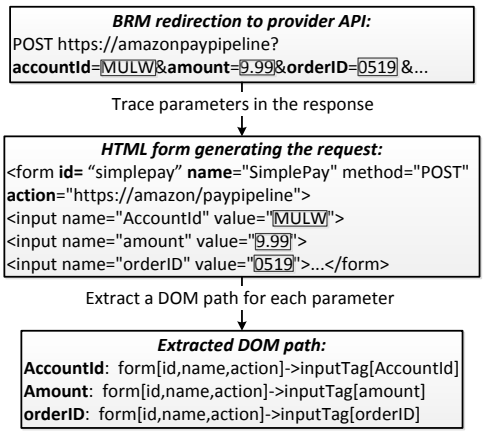

Figure 8. DOM path of HTML form elements extraction.

Occasionally, elements of interest are communicated in a customized format. For example, the content of `openid.ext1.required` used by Google ID (see Section II-B) is formulated in a nonstandard way: `(email,firstname,lastname,language)`. To extract these elements, the collector analyzes over-length parameters it finds, which in our prototype are those including more than 15 characters, and further partitions the parameters using common delimiters such as comma and period. What can also happen, though very rarely, is that a parameter is encoded, which in all the cases we encountered is through Base64. Our approach detects the presence of encoding when an over-length parameter is found to contain no delimiters. In this case, we try to decode it using common schemes, Base64 in particular, and declare a success when delimiters show up and elements extracted are included in some invariants. When this attempt fails, human intervention is needed to supply our system with the format information, which the provider must well specify to make integration possible. Actually, this never happened during our study on real-world integrations (Section IV-B).

C. Invariant Analysis

Invariant detection. A critical step to secure an integration is to identify the invariant relations among the messages produced by its normal operations. We expect that such invariants can help us unambiguously determine whether an inbound message is a legitimate one, belonging to one of the integrator's ongoing transactions. To this end, our approach is designed to detect four types of invariants:

local invariants for checking the validity of a transaction message given prior messages in the same transaction, *transaction-specific invariants* for connecting a message to a transaction, *integrator-specific invariants* for detecting the messages unrelated to the integrator under protection, and the other invariants for characterizing the service provider and determining when a transaction starts and when it ends. These invariants are identified by running a set of differential analyses on the four traces collected, as elaborated below.

• *Local invariants*. Given a trace, a *local element invariant* L is a set of elements, each from a different RT of a transaction, with the following property: for any two elements e, $e' \in L$ and their value attribute v (denoted by $e.v$), we have $e.v = e'.v$, that is, they all have the same value. For example, the `amount` element extracted from RT1.response in Figure 7 (the item's price) matches the `gross` element of RT3.request. Such an invariant relation should always be kept throughout a transaction to ensure the consistency in its execution.

To detect all such invariants on a trace T_1, the policy generator inspects each RT V in the transaction and compares the value of its element e (i.e., $e \in V$) with all the elements in the RT's predecessors, until one of them V' is found to include an element e' such that $e'.v = e.v$. When this happens, we first check whether e' is already in a local invariant set: if so, e is added to that set; otherwise, a new set is created to contain e' and e. In our running example in Figure 7, InteGuard compares the value of $gross$ in RT3 ($RT3.gross.v$) with those of the elements in RT1, which reveals the invariant $RT3.gross.v = RT1.amount.v$. Note that for local invariants, we just look at the relations between elements, instead of their specific values (the content of $amount.v$ for example). After generating all the invariants for T_1, our approach further compares them with those detected from T_2, the other trace in the same group, to update the content of individual invariant sets: for any invariant L of T_1, we update it as $L \leftarrow L \cap L'$ if $L \cap L' \neq \phi$, where L' is an invariant for T_2; L is dropped if such L' cannot be found. Intuitively, only the relations held in both traces (whose input parameters for "attributes of interest" are completely different) are considered to be true invariants. This treatment helps reduce false positives. Another false-positive control we took is to drop all detected invariants associated with the elements whose values are below three bytes, such as `returnFlag` in RT1.response (Figure 7).

Another local invariant InteGuard utilizes is the sequence of web APIs invoked during a transaction and the features of their responses. Different from the call sequence of a desktop program, which has long been used for anomaly detection [19], [20], a web API sequence includes the functions on both the integrator and the provider sides. Therefore, a call in the sequence is described not only by its API name but also by its domain. For example, in our running example (Figure 7), the calls the integrator can observe are (https://jeff.com/placeOrder.aspx, https://jeff.com/finishOrder.aspx). Also used in the sequence are the API parameters and HTTP response status code (e.g., 302, 200, etc.) of the call's response. This invariant works particularly well on web API integration, as it operates pretty mechanically: for legitimate transactions completed through the same integration, their API call sequences never change.

• *Transaction-specific invariants*. A real-world integrator works concurrently on multiple transactions. Also, the traffic it generates can be mixed with that produced by other activities unrelated to any transaction. As an example, consider a web store: at any given moment, it may process many checkout transactions and also serve the shoppers who are just browsing its web pages and choosing commodities. A challenge here is how to determine whether an inbound HTTP message is related to checkout, and if so, which transaction it belongs to. Our approach addresses this issue by identifying the invariants specific to a transaction.

Actually, whether an inbound HTTP request is related to a transaction can be easily determined by looking at the API function it calls: if the API is transaction-related (like https://jeff.com/placeOrder.aspx), as indicated by the web API sequence discussed above, then the message belongs to some transaction. Finding out which transaction it should go to is not hard either when the message comes from the client, whose session ID serves as a transaction-specific invariant. What gives us trouble is the messages exchanged directly between the integrator and the provider, which do not share the session ID with the RTs from the client, even when they are actually within the same transaction.

To link such messages to their transactions, we resort to a subset of local element invariants. The idea is to find those local invariants that uniquely identify a transaction. Denote a trace T's element e by $T.e$. Given all four test traces (T_1, T_2) from Group 1 and (T_1', T_2') from Group 2, a local invariant L is also transaction-specific if for any $e \in L$, $T_1.e.v \neq T_2.e.v \neq T_1'.e.v \neq T_2'.e.v$. In other words, every element in this invariant has a different value in all four test transactions. Such an invariant can be used to identify the transaction a message is associated with, because of the following reasoning. T_1 and T_1' share transaction parameters except integrator identity. Therefore, the invariant with different values in these traces should be related to either integrators or their transactions. The former is ruled out by the comparison between T_1 and T_2, which share the integrator. Then, we can conclude that what are left must be specific to transactions. Examples of such invariants include order/transaction ID. To the integrator, such invariants appear on both the messages from client and those from the provider, and thus can be used together to link these messages to a specific transaction.

• *Integrator-specific invariants*. With the extensive use of

open redirects such as `returnURL` in Figure 3, the integrator needs to know whether it is the right recipient of an inbound message. Our solution, again, is a simple differential analysis that compares different test traces collected. Specifically, given an element e, it is integrator-specific if the following two conditions are met: for the four test traces (T_1, T_2) and (T_1', T_2'), (1) $T_1.e.v = T_2.e.v$ and $T_1'.e.v = T_2'.e.v$, and (2) $T_1.e.v \neq T_1'.e.v$. Since T_1/T_1' (and also T_2/T_2') come from the transactions that differ *only* in the integrator's identity, the elements that hold the same values (see (1)) within the same group, that is, associated with the same integrator, and different values in different groups (see (2)) should be identity-related. An example is `payee` in our running example (Figure 7). This approach automatically removes the invariants related to transactions, which must have different values for the transactions from the same integrator, and also suppresses other false positives caused by the parameters missed on the list of attributes of interest: for example, elements appended by the provider to the message the client sends to the integrator, such as `payMethod`, will not be picked out for integrator identification, when it holds the same value `creditCard` on all four traces. The invariants identified in this way are used by the ICAP server to catch the transaction messages the integrator is not supposed to receive.

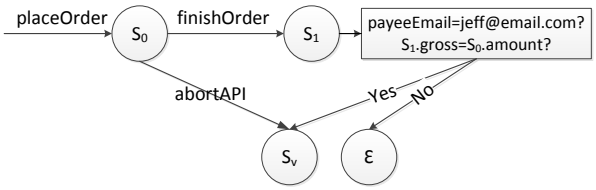

Figure 9. FSM policies for NopCommerce's integration of Amazon Simple Pay. Inside the rectangle box are the security checks performed in State S_1. The FSM is triggered by the function call `PlaceOrder` (RT1.request in the running example)

• *Other invariants*. Since an API integration is only part of a hybrid web application (e.g., the checkout integration in a web-store system), we need to find out when InteGuard's protection should be activated and when it should be turned off during the application's operation. In practice, the start of a transaction can always be identified from the first API call the application makes to the service provider's website, which always goes through an HTTP response to the client (i.e, a BRM). When this happen, the ICAP server also picks up the HTTP request, which is temporarily buffered, as part of the transaction traffic.

The completion of a transaction can be identified from the API sequence of the integration. In the case that a client aborts a transaction through the links provided by the integration, we can compare the abort trace with its counterpart in Group 1 to identify their difference in API sequences and parameters, which serves as an invariant for determining a normal termination of a transaction. When the client simply browses away from transaction-related web pages or closes related windows/tabs, we have to resort to other information to find out what has happened. Particularly, the client in the middle of a transaction is not supposed to alter the state of the integrator through the request unrelated to the transaction. Such an attempt is detected by our approach from a `POST` request (unrelated to any transaction) produced by the client whose session ID is associated with an ongoing transaction. This combination, i.e., non-transaction `POST` and transaction-related session ID, serves as an invariant that InteGuard uses to detect the end of a transaction.

Finally, we also need to consider the invariants that characterize the service provider. What is used in our system for this purpose are the provider's domain names: if a provider-domain element appears on a given RT across all the traces collected, InteGuard picks it up and treat it as a provider-related invariant.

Security policy generation. Using the invariants detected, our approach further builds a set of security polices for integration protection. These policies include a *global* policy utilized by the ICAP server to quickly filter incoming messages before an in-depth invariant check happens. Such a policy requires the server to first look at a message's URL: when transaction-related APIs are not found there and the message also does not meet the termination condition stated above, it is ignored.

After a message passes the global policy check, its content is further inspected using different types of invariants. Security policies built upon these invariants are organized as an FSM for each integration. Specifically, such an FSM can be described by a tuple $(S, \Sigma^+, \Sigma^-, \delta, S^+, \delta^+, S^-, \delta^-)$. Here, S is a set of states, each of which contains a set of local invariants (including transaction-specific ones) and integrator-specific invariants to be checked at the state. These states include an initial state s_0, where a transaction starts, a transaction completion state ν and an error state ϵ. At ϵ, InteGuard can accommodate customized polices, ranging from raising an alarm to stopping the whole transaction. Σ^+ is a set of transaction-related request messages, which are paired with their corresponding responses in Σ^-. Each request/response pair, i.e., a round-trip, triggers a state transition, according to the function δ: $S \times \Sigma^+ \times \Sigma^- \rightarrow S \cup S^+ \cup S^-$, where S^+ and S^- will be explained later. Actually, δ models the operations each state performs in response to an HTTP RT, which includes extracting and checking its elements according to the invariants, and updating the content of the invariant set to prepare for the inspection at the next state. An inconsistency between the value of any invariant and its corresponding element on the message constitutes a policy violation, which moves the state to ϵ. Also important here is determining whether the messages are supposed to be received at the current state, based

upon transaction-specific and integrator-specific invariants. An incoming message not associating with any FSM and the current integrator will cause the ICAP server to raise an alarm. As an example, Figure 9 shows the FSM of our running example (Figure 7): at State S_1, a local invariant S_1`.gross = `S_0`.amount` and an integrator-specific invariant $payeeEmail =$`jeff@email.com` are verified.

In addition to regular states in S, the FSM also includes special states in S^+ and S^-, which work on a single message (a request or a response) instead of a round trip. This happens when the integrator receives an HTTP request from the client and needs to retrieve information from the service provider (through a few RTs) before sending back a response. An example is Interspire's integration of PayPal Express (Figure 10). A checkout through the integration requires the merchant to contact PayPal for a client's payment information between an RT initiated by the client. Such an operation is modeled by states in S^+, whose state transitions are governed by a different function δ^+: $S^+ \times \Sigma^+ \rightarrow S$. Invariant checking at such a state only happens to an HTTP request. After the invariant set has been updated, the system moves into a new state in S, which processes the RT (e.g., the one that queries PayPal) it receives. Following such operations is a state in S^-, which only accepts the response corresponding to a prior pending request, through the function δ^-: $S^- \times \Sigma^- \rightarrow S$. In this way, the system gets back to a regular state.

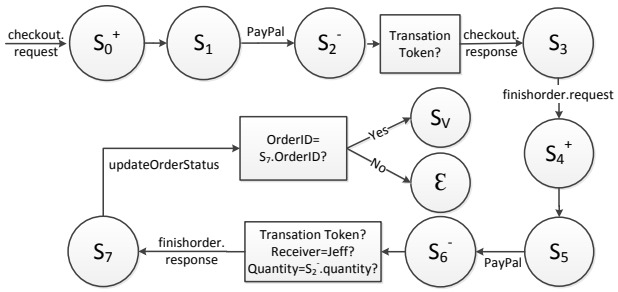

Figure 10. FSM policies for Interspire's integration of PayPal Express.

To build such an FSM, the policy generator goes through a RT sequence recorded in the trace: starting from the first one, each RT results in a state that records the invariants expected from the next RT. The special states are detected from the order in which a request, its response and the RTs in-between are observed by the collector. At these states, transitions are triggered by a single message. All the states except S_0 are connected to ϵ, the error state, when they receive messages that do not meet any one of the invariants expected at these states.

D. Online Protection

Server configuration. The Squid proxy utilized by InteGuard holds the integrator's domain name. The client that accesses a hybrid web application in the domain sends HTTP requests to the proxy, which rewrites their URLs to forward the request (through the ICAP server) to an internal website that provides the real service. In this way, the web traffic produced by the application always goes through the proxy. This configuration, however, could cause problems for some integrations, particularly when a service provider's signature on the integrator's URL needs to be verified by the website. The trouble here is that the website can use a different domain or a sub-domain than that of the proxy. This causes the verification to fail when the commercial off-the-shelf application is used and change of its code is not an option. Our solution here is letting the proxy keep the proxy's domain within the `Host` field of the HTTP messages it forwards to the website. Since the application gets the domain name (for the signed URL) from that field, the signature can then be successfully verified.

Policy enforcement. During its runtime, the proxy intercepts the website's HTTP traffic and then passes the messages to the ICAP server, which was built upon a GreasySpoon service in our prototype. On messages related to ongoing transactions, the ICAP server performs the global-policy check, extracts elements of interests from the messages and checks their compliance with security policies. Element extraction here was implemented using Java SDKs and the compliance check is performed by a GreasySpoon script that takes the descriptions of FSMs as its input. The state of a transaction can be kept in memory or a database associated with the ICAP server, in the same way that a web store handles pending orders. Such policies can be adjusted when false positives are discovered during InteGuard's operation.

IV. Evaluation

We performed a thorough evaluation of our InteGuard prototype to understand its effectiveness in protecting vulnerable integrations, the false positives its incurs and its performance impacts. Our study shows that our approach indeed achieves practical integration protection: it defeated complicated exploits on subtle logic flaws within real-world integrations, at a zero false-positive rate and small performance overhead. Here we elaborate this study.

A. Experiment Settings

Integrations. We used the hybrid web applications with known integration vulnerabilities in our study. These web applications include leading merchant systems and popular web services, all integrated with the APIs provided by well-regarded payment or SSO service providers. More specifically, we tested our techniques over 6 vulnerable checkout integrations within Interspire (Starter Edition 5.5.4) and NopCommerce (1.60), involving PayPal, Amazon Payments and Google Checkout, as reported in our prior research [1], as well as 5 recently discovered faulty SSO integrations [2], involving popular websites such as sears.com, janrain.com,

Table I
EFFECTIVENESS EVALUATION

No.	Application	Service Integrated	Invariant Type	Invariant Violated	Detected
1	NopCommerce	Paypal Web Payment Std	Local	paymentAmount == transactionAmount	Yes
2	NopCommerce	Amazon Simple Pay	Integrator-Specific	recipientEmail == constant value	Yes
3	NopCommerce	Amazon Simple Pay	Integrator-Specific	certificateUrl domain == provider domain	Yes
4	Interspire	Paypal Web Payment Std	Transaction-specific	transactionId == custemField	Yes
5	Interspire	Paypal Express	Local	signedOrderId == id	Yes
6	Interspire	Google Checkout	Web API Sequence	API seqence	Yes
7	Smartsheet.com	Google ID	Local	signedList == signedList	Yes
8	Janrain	Google ID	Local	discoverytoken == discoverytoken	Yes
9	Sears.com	Facebook SSO	Integrator-Specific	returnURL == constant value	Yes(simulated)
10	Shopgecko.com	Paypal Access	Local	openid.type == type.value	Yes
11	FarmVille	Facebook SSO	N/A	N/A	No

etc., and famous IdPs like Google, Facebook and PayPal. The details of these integrations are presented in Table I.

Our evaluation also utilized carefully-crafted exploits on 11 logic flaws within these integrations, as reported in our prior work [1], [2]. Since all vulnerable SSO integrations were already fixed, we had to work on the traces recorded from our prior attacks.

System settings. Our proxy and ICAP server were installed on a system with QuadCore i5-2400 3.10GHz CPU and 8 GB memory. The ICAP server was running inside a Java Virtual Machine with 5 GB memory. The system was equipped with Linux 2.6.32. The web server that powered merchant applications was on a system with Core 2 1.83GHz CPU and 4 GB memory. Its operating system was Windows Server 2008 R2.

B. Effectiveness

Our evaluation was performed on all vulnerable integrations reported in two recent papers [1], [2], except those in the following categories: (1) proprietary web services with direct integrator-to-provider communication, (2) provider-side flaws, (3) non-transaction flaws and (4) response failure. For (1), we did not have access to the merchant applications used by Buy.com and JR.com, and their communication with CaaS providers, which makes it impossible to build their FSM policies. An example for (2) is the Flash cross-domain problem in Facebook Access [2]. The flaw was introduced by Facebook, allowing the adversary to steal the integrator's secret access token even without interacting with it at all. In (3) is Freelancer.com, whose flaw needs to be exploited before an SSO transaction happens [2]. A case in (4) is the Facebook integration within zoho.com: apparently, it detects invariant violations but takes a problematic response (redirecting a client to a suspicious web site) that leaks out the client's login credential through a referral header [2]. Note that the flaws in (2), (3) and (4) seem to be less pervasive than the vulnerabilities caused by the integrator's failure in ensuring security-critical invariants during a transaction (all CaaS flaws discovered [1] and 5 in 8 SSO flaws [2]), which our approach is designed to fix.

We installed Interspire 5.5.4 and NopCommerce 1.60 on our local system and generated the security policies (built upon all types of invariants) for their integrations using the traces collected from normal checkout transactions. The list of attributes of interest used for generating those traces includes a shopper' user account with the merchant, the name/price/quantity of the commodity she chooses, delivery method and the shopper's account with the CaaS providers (Amazon, PayPal and Google). Note that we did not mean to make the list complete, in order to understand the false positives our approach could cause when the diversity of the traces is limited. The traces for SSO integrations were gathered from real login processes performed on their web sites, using different user accounts on both the integrator and the service provider sides (attributes of interest). These traces were sufficient for constructing security policies, as SSO logins do not have direct integrator-to-provider communication. The problem is that we could not change these real web sites' accounts with their SSO providers and thus all four traces we collected came from the same integrator. As a result, the FSMs generated for their integrations only included local and transaction-specific invariants.

Results. We ran all the exploits reported in the prior research and a new one we discovered on Interpire and NopCommerce, and found that InteGuard defeated all of them. For SSO integrations, we replayed 5 attack traces [2] against our prototype, which caught 3 of them. The cases our approach missed include the Facebook Access integrations in sears.com and Farmville's integration of Facebook Legacy Canvas Auth [21]. The problem with sears.com is related with an integrator-specific invariant, which we did not get the traces to detect. However, we manually analyzed the traces we had and found that once we can test its integration system using a different integrator identity (its domain name in this case), our prototype can generate the right invariant to defeat the exploit. The logic flaw in FarmVille's integration, apparently, is caused by its failing to verify Facebook's signature [2]. The invariant relation between a signature and the signed content cannot be checked without semantic information like signing algorithms, key, etc., which invariant-

based approaches cannot handle. Fortunately, such a type of flaws is rare: this is the only documented case in which signature verification is completely missing.

Table I elaborates the outcomes of the effectiveness evaluation. Here we describe some examples. The details of the rest of the exploits can be found in [1], [2].

CaaS integrations. For NopCommerce's integration of Amazon Simple Pay, as described in Section II-B, our prototype identified `payeeEmail` in BRM2 as an integrator-specific invariant. This invariant was used by InteGuard to detect the exploit that causes Amazon to send the notification for Mark's transaction to Jeff (see Section II-B): the `payeeEmail` field in BRM2 was found to be inconsistent with the invariant, which should be Jeff's email.

As another example, consider Interspire's integration of PayPal Express, as illustrated in Figure 11. After a client initiates a checkout transaction through an HTTP request, the integrator first uses RT1 to acquire a transaction token (`token`) from PayPal, and then uses BRM1 to take the client to PayPal to pay for the order generated for the transaction, using the token as the transaction's identifier. After the payment is done, PayPal uses BRM2 to bring the client back to the integrator and call the merchant's API `finishOrder`. The API performs RT3 to get the payment details from the PayPal and then redirects the client to call another merchant's API `updateOrderStatus` with signed `orderID`. That API sets the status of that order to "PAID".

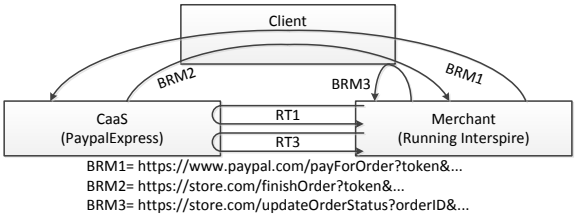

Figure 11. Interspire's integration of PayPal Express.

A logic flaw revealed by our prior research [1] is that a malicious client can replace the order ID in one transaction with that in a different transaction in the BRM that calls `updateOrderStatus`. The function will stamp the second order as "PAID", even when it is not, as long as the first one is paid. As a result, a shopper can pay for a cheap order (the first one) and get an expensive item from another order (the second one). A new finding we made in our study is that the token of an unpaid transaction can also be swaped with that of a paid transaction. When this happens, the integration code mistakenly associates the paid transaction token with the unpaid order and therefore allows the malicious shopper to get one item at another item's (lower) price.

Both exploits were defeated by InteGuard, which identified both `token` and `orderID` as local invariants. The attack messages for supplanting these elements bear a session ID, also a transaction-specific invariant, matching that of one transaction but `token` or `orderID` violates the local invariant of it. This makes the system raise an alarm.

SSO integrations. Our approach also caught both exploits described in Section II-B. For the Google ID integration in smartsheet.com, InteGuard parsed the content of `openid.ext1.required` (integrator's response in BRM1) and `openid.signed` (Google's response in BRM2). The elements extracted from the first list should match these on the second list, which was detected as a set of local invariants. During the attack, one of such invariants was found to be violated: that is, the email element on the first list, which smartsheet.com asked Google to check, could not be found on the second list, which was Google's response. This revealed the malicious attempt made by the client. Similarly, for the PayPal Access integration in Shopgecko.com, a local element invariant found was that `type.email` in BRM1 (Shopgecko's request to PayPal) should match the content of the same field in BRM2 (PayPal's reply to Shopgecko). This invariant was found to be broken during the attack, in which the client changed the field in BRM1 to the type of mailing address, thus causing inconsistency with what the proxy saw in the response part of the BRM, i.e., the type of PayPal ID.

C. False Positives

We studied the level of false positives the invariants identified by InteGuard could incur. For each CaaS integration, we conducted 100 to 300 checkouts through their PayPal, Amazon and Google service providers to "buy" commodities from our web stores powered by Interspire and NopCommerce. To make our experiments as realistic as possible, we had 5 to 20 different users[4] check out different items simultaneously from our stores. These users' profile information (names, address, and others) was set differently. During our experiments, they randomly chose commodities with parameters of individual items (e.g., quantities, sizes of shoes, disk space, etc.) randomly set according to the constraints imposed by user interfaces. Those checkouts were paid at PayPal, Amazon and Google. Also, we simulated random user behavior, such as clicking on "Back" button, leaving merchant or CaaS websites and later returning through old URLs, randomly crawling URLs on merchant web pages during transactions, etc., in an attempt to generate noise traffic to confuse InteGuard. For each SSO integration, we conducted 20 logins to their corresponding websites through their Facebook, Google and PayPal IdPs. Each of

[4]20 user accounts were opened for the study through PayPal, Amazon and Google's Sandbox development environment [15], [22], [23] and used for evaluating the integrations except Interspire's PayPal Payments Standard with Instant Payment Notification (IPN), which is not well supported here by Sandbox. We had to evaluate this integration using 5 real accounts.

Table II
PERFORMANCE EVALUATION

BRM / Channels	32 Users (Delay in **ms**)			256 Users (Delay in **ms**)		
	W/O InteGuard	**W/ InteGuard**	**Overhead (%)**	**W/O InteGuard**	**W/ InteGuard**	**Overhead (%)**
HTML Form(Interspire)	4331	4475	3.32%	19985	20305	1.60%
HTTP 3XX(NopCommerce)	684	687	0.44%	1093	1101	0.73%
JavaScript(Synthesized)	1322	1352	2.27%	9585	9658	0.76%
JSON(Synthesized)	473	475	0.42%	3967	4001	0.86%
Meta Refresh(Synthesized)	4848	4904	1.16%	26061	26920	3.30%

these logins involved a different user with different profile. The HTTP traces recorded from the SSO transactions were replayed to our prototype.

Throughout 1,000 real transactions (100 SSO logins and 900 checkouts), InteGuard did not trigger any false alarms. We believe that this zero-false-positive result comes from the nature of web API integrations. In addition to the information for identifying transactions and integrators, other variations in the traffic produced by these integrations across different transactions mainly come from user inputs. In our running example, what causes the dynamics over the same integrator's different transactions is `amount`, which changes with different items a shopper chooses (for more complicated examples, see the traces here [16]); other elements like `orderID`, `AccountId`, `payeeEmail` always stick to transactions or integrators. As a result, a small set of traces generated using different user inputs (attributes of interest), different transactions and different integrators can be sufficiently diverse for avoiding false invariants.

D. Performance

We also put our technique to a test to understand its performance impact, which mainly comes from parsing different types of web content to extract elements. In our study, we selected representative integrations from different content categories, including HTTP 302 redirection, HTML form, meta refresh, JavaScript and JSON. The details of these integrations are listed in Table II. In real world, JSON and meta refresh are used in Buy.com and Janrain.com respectively. Also, Amazon SDK includes the support for JavaScript-based BRM. We synthesized the integrations in these categories, as we did not have access to real-world websites' integration code.

We ran Apache JMeter [14] to generate test traffic. Specifically, we first recorded the HTTP trace of a manual checkout process for each integration, and then used JMeter to replay the trace, with its parameters dynamically updated by scripts. In this way, we generated workload for our prototype that protects each of these integrations under the scenarios of 32 concurrent checkout transactions and 256 concurrent checkouts, through 5 user accounts on the CaaS sides. Note that 256 is Apache's default setting of maximum number of concurrent connections [24]. In our experiment, all except 5 of these transactions did not run to completion, as PayPal and other CaaS providers do not handle two orders from the same user at the same time. Nevertheless, these transactions forced our system to parse their web content for invoking provider APIs and perform initial invariant checks, which constitutes the major portion of the transaction workload. The average time of a RT in these transactions (including the 5 completed ones) was recorded and compared with the delay observed from a baseline setting that only involved a standard web proxy without any InteGuard components (which are in the ICAP server). We did not take the proxy out of the baseline because it does nothing but forward traffic to the ICAP server, and for the commercial websites capable of handling a large number of transactions, they need a load balancer that plays the same role as the proxy here to distribute web traffic to the servers in their web farms.

The experimental results are presented in Table II, which shows that the overhead incurred by InteGuard are negligible (no more than 3.32%) even in the presence of 256 concurrent transactions. Although the ICAP server undertook the tasks such as parsing JavaScript, HTML pages, etc., such web content was typically small in an integration-related transaction (e.g., a few hundred bytes of JavaScript code, see Page 7) and did not add much to the overall performance overhead. Our FSM polices did not bring in much memory burden either: the memory consumption of the ICAP server stayed constant (about 1250MB) when the number of the concurrent transactions grew from 32 to 256; also, the presence and the absence of the FSM policies only changed the memory usage marginally (about 150 MB out of the 1250 MB memory consumed by the ICAP server even without the FSM policies). Note that this level of workload (256 concurrent connections) already hit the performance limit of a single Apache server the merchant store ran upon (a server farm is needed for a big store to process tens of thousands of transactions), but not that of the ICAP server operating InteGuard. Actually, we found that in some cases, the delay caused by our security check was so small that it was completely overshadowed by the variations in network communication latency, forcing us to repeat experiments 10 to 30 times to get an averaged result. This gives us reason to believe that with a proper level of hardware support, InteGuard should be able to handle a large

number of transactions a big web store receives.

V. Discussion

Our simple invariant identification technique turns out to be very effective (Section IV), because web APIs (payment, SSO) often have a relatively small set of input parameters, most of which are expected to be set by the user, and can therefore be retrieved from user interfaces and covered by the list of attributes of interest. This helps achieve diversity in the inputs and reduces false positives. Also, the integrator and the provider typically do not perform complicated transformations on their inputs to produce responses. As a result, most invariants (e.g., orderID, price, etc.) are rather simple (exact match) and can be found from web traffic. On the other hand, our design cannot handle complicated invariant relations, such as that between signed content and its signature (Section IV-B). New approaches that address this problem should be developed to improve our techniques. Also, although InteGuard is for exploit prevention, the invariants produced thereby can help with vulnerabilities detection within an integration[5]. These directions are definitely worth further research effort.

In our evaluation study, we did not observe any false alarms. This, however, does not mean that our approach is completely immune to false positives. Actually, although the parameters of transactions set by the service provider for a specific integration are largely stable, some of them could change over a relatively long period of time. When this happens, InteGuard will falsely report a foul play on a legitimate transaction. A better understanding of this issue and development of effective mitigation when necessary need to be studied in the follow-up research.

Our performance evaluation (Section IV-D) provides preliminary evidence that InteGuard works efficiently under heavy workload. On the other hand, further studies are still necessary to understand how our approach performs in the presence of an even larger number of transactions. In this case, we need to move our system onto a server farm, for example, those deployed on public commercial clouds like Amazon EC2, under different load balancing strategies. Also, the policy checking step indeed introduces some additional states for each transaction, which theoretically make the system more vulnerable to a distributed denial of service (DDoS) attack. On the other hand, the problem exists in any corporate networks that utilize ICAP and other content inspection tools for the purposes such as malware detection, and our preliminary results show that our approach only increases CPU/memory overheads marginally compared with those of the ICAP server itself. This indicates that use of InteGuard should not raise such a DDoS risk significantly, though further studies are still needed to better understand the issue.

Finally, InteGuard is designed for protecting multi-party web applications in the absence of their source code and semantics (particularly the necessary conditions for an SSO or payment transaction to succeed, which needs to be manually specified). When such information is accessible, techniques for code analysis and vulnerability detection can be built here. We are happy to see that progress has been made on this direction, with recent work [25] that extracts specifications from the implementation of web applications for detection of their security flaws.

[5]A caveat here is that violation of those invariants may not directly lead to an exploit, which needs semantic information to identify.

VI. Related Work

Techniques for avoiding web-service logic flaws or detecting exploits of such flaws have become a new line of research in web security in recent years. Swift [26] and Ripley [27] are compiler techniques to help web developers securely split or duplicate a web program's logic between the client (which is assumed to be untrusted) and the server (which is trusted). The goal is to make sure that either security-critical logic is performed on the server or security-critical results produced by the client are re-examined on the server. In addition to these secure-by-construction approaches, effort has also been made to address logic flaws within existing web applications. Most prior research in this area focuses on automated analysis of a web application's source code. Examples include Swaddler [28], MiMoSA [29], Waler [4], RoleCast [30], the study on Execution After Redirect [31], SAFERPHP [32], WAPTEC [33], a role-based static analysis proposed in [34], APP_LogGIC [35] and recently Fix-Me-Up [36]. Particularly, Waler [4] utilizes Daikon [37] to generate likely-invariants of web programs and then runs model checking to identify program paths that could lead to violation of these invariants. Attempts have also been made to detect logic flaws in the absence of source code. A prominent example is NoTamper [38], a black-box technique that automatically generates inputs to test the server-side logic of a web application. The idea is to detect the existence of the input the server-side program accepts while the client-side validation logic does not. Most related to our work is BLOCK [5], which also extracts invariants from web traffic and works within a proxy. BLOCK leverages simple invariants such as session ID and the order of calls to detect three types of exploits: skipping the authentication step, manipulating HTTP parameters, and altering the normal work-flow expected by the server. It does not include a false positive control.

More generally, there is a line of research on web application firewall [39], [40] that controls the erroneous behavior of the application under protection and prevents exploitation of its vulnerabilities. This firewall framework, at the high level, can accommodate the new techniques for defending

against the attacks on logic flaws, such as NoTamper, BLOCK and our approach. On the other hand, the detailed designs of these defense engines are nontrivial, which are based upon the unique features of the applications and the types of the flaws in concern. Particularly, InteGuard works on multi-party web applications, a research perspective that has never been explored before.

As mentioned above, all prior work focus on conventional two-party web applications (including a client-side component and a server-side component), while InteGuard is designed to secure the integration of multiple web services: a client needs to interact with at least an integrator and a service provider during a web application's operation. This computing paradigm is complicated and error-prone, as discovered in our recent studies [1], [2], which brings in new security challenges: particularly, our approach needs to identify the communication among these three parties that belongs to the same transaction and ensure the consistency in the content of the messages exchanged directly or indirectly between them, when the interactions between the client and the service provider are not observable to our proxy.

VII. Conclusion

In this paper, we present InteGuard, the first proposal that protects vulnerable integrations of third-party web services. InteGuard uses a set of traffic invariants to check the HTTP messages during a transaction to detect exploit attempts on its logic flaws. These invariants are acquired through a suite of new techniques designed to address the unique challenges service integrations pose. Our evaluation shows that InteGuard defeated real exploits on the subtle logic flaws within leading web services at a small performance expense.

We plan to make available the source code of InteGuard in the near future and further improve its design. More generally, InteGuard is just a first step toward secure web service integration, offering only limited protection: for example, it does not protect provider-side flaws. Given the importance of the problem, we expect further effort to be made in this direction.

Acknowledgements

This work is supported in part by the National Science Foundation under Grant CNS-1117106 and CNS-1017782.

References

[1] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online - security analysis of cashier-as-a-service based web stores," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy*. Oakland, CA: IEEE, 05/2011 2011.

[2] 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 *Proceedings of the 33nd IEEE Symposium on Security and Privacy*. San Francisco, CA: IEEE, 05/2012 2012.

[3] "Unpredictable Domain Communication," http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/LocalConnection.html.

[4] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *Proceedings of the USENIX Security Symposium*, Washington, DC, August 2010.

[5] X. Li and Y. Xue, "Block: a black-box approach for detection of state violation attacks towards web applications," in *ACSAC*, 2011, pp. 247–256.

[6] "Janrain," http://janrain.com/.

[7] "Choose the Best Open Source CMS for 2010," http://news.softpedia.com/news/Choose-the-Best-Open-Source-CMS-for-2010-158440.shtml.

[8] "OpenID 2009 Year in Review," http://openid.net/2009/12/16/openid-2009-year-in-review/.

[9] "Security advisory to websites using OpenID Attribute Exchange," http://googlecode.blogspot.com/2011/05/security-advisory-to-websites-using.html.

[10] "OpenID Foundation security advisory," http://openid.net/2011/05/05/attribute-exchange-security-alert/.

[11] "PayPal OpenID Implementation Details," https://www.x.com/developers/community/blogs/itickr/paypal-openid-implementation-details.

[12] "Squid," http://www.squid-cache.org/.

[13] "GreasySpoon," http://greasyspoon.sourceforge.net/.

[14] "Apache JMeter," http://jmeter.apache.org/.

[15] "Paypal Sandbox," https://developer.paypal.com/.

[16] "Supporting Materials," https://sites.google.com/site/InteGuard.

[17] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: automatic protocol reverse engineering from network traces," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362917

[18] "Using native JSON," https://developer.mozilla.org/En/Using_native_JSON.

[19] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, pp. 151–180, August 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298081.1298084

[20] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE conference on Security and privacy*, ser. SP'96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–128. [Online]. Available: http://dl.acm.org/citation.cfm?id=1947337.1947356

[21] "Legacy Canvas Auth," http://developers.facebook.com/docs/authentication/fb_sig/.

[22] "Google Sandbox," http://support.google.com/checkout/sell/bin/answer.py?hl=en&answer=134469&topic=12157&ctx=topic.

[23] "Amazon Sandbox," https://payments-sandbox.amazon.com/sdui/sdui/index.htm.

[24] "Apache MPM Common Directives," http://httpd.apache.org/docs/2.0/mod/mpm_common.html#maxclients.

[25] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations," in *Proceedings of 20th Annual Network & Distributed System Security Symposium*, 2013.

[26] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 31–44, October 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294265

[27] K. Vikram, A. Prateek, and B. Livshits, "Ripley: Automatically securing distributed Web applications through replicated execution." in *Conference on Computer and Communications Security*, Oct. 2009.

[28] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Gold Coast, Australia, September 2007, pp. 63–86.

[29] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna, "Multi-Module Vulnerability Analysis of Web-based Applications," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October 2007, pp. 25–35.

[30] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: finding missing security checks when you do not know what checks are," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 1069–1084. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048146

[31] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 251–262. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046736

[32] S. Son and V. Shmatikov, "Saferphp: Finding semantic vulnerabilities in php applications," in *Proceedings of the ACM SIGPLAN Sixth Workshop on Programming Languages and Analysis for Security (PLAS 2011)*, San Jose, CA, 2011.

[33] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 575–586. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046774

[34] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028078

[35] G. Stergiopoulos, B. Tsoumas, and D. Gritzalis, "Hunting application-level logical errors," in *ESSoS*, 2012, pp. 135–142.

[36] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications," in *Proceedings of 20th Annual Network & Distributed System Security Symposium*, 2013.

[37] "Daikon," http://groups.csail.mit.edu/pag/daikon/.

[38] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications," in *CCS'10: Proceedings of the 17th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2010.

[39] D. Ingham, O. Rees, and A. Norman, "Corba transactions through firewalls," in *Proceedings of the International Symposium on Distributed Objects and Applications*, ser. DOA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 284–. [Online]. Available: http://dl.acm.org/citation.cfm?id=519622.790762

[40] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, "Bridging the gap between web application firewalls and web applications," in *Proceedings of the fourth ACM workshop on Formal methods in security*, ser. FMSE '06. New York, NY, USA: ACM, 2006, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/1180337.1180344