Securing Multiparty Online Services via Certification of Symbolic Transactions

Eric Y. Chen
Carnegie Mellon University
Moffett Field, CA, USA
eric.chen@sv.cmu.edu

Shuo Chen, Shaz Qadeer, Rui Wang
Microsoft Research
Redmond, WA, USA
{shuochen, qadeer, ruiwan}@microsoft.com

Abstract— **The prevalence of security flaws in multiparty online services (e.g., single-sign-on, third-party payment, etc.) calls for rigorous engineering supported by formal program verification. However, the adoption of program verification faces several hurdles in the real world: how to formally specify logic properties given that protocol specifications are often informal and vague; how to precisely model the attacker and the runtime platform; how to deal with the unbounded set of all potential transactions.**

We introduce Certification of Symbolic Transaction (CST), an approach to significantly lower these hurdles. CST tries to verify a protocol-independent safety property jointly defined over all parties, thus avoids the burden of individually specifying every party's property for every protocol; CST invokes static verification at runtime, i.e., it symbolically verifies every transaction on-the-fly, and thus (1) avoids the burden of modeling the attacker and the runtime platform, (2) reduces the proof obligation from considering all possible transactions to considering only the one at hand.

We have applied CST on five commercially deployed applications, and show that, with only tens (or 100+) of lines of code changes per party, the original implementations are enhanced to achieve the objective of CST. Our security analysis shows that 12 out of 14 logic flaws reported in the literature will be prevented by CST. We also stress-tested CST by building a gambling system integrating four different services, for which there is no existing protocol to follow. Because transactions are symbolic and cacheable, CST has near-zero amortized runtime overhead. We make the source code of these implementations public, which are ready to be deployed for real-world uses.

Keywords-multiparty protocol; symbolic transaction; CST; verification; online payment; single-sign-on

I. INTRODUCTION

Modern web applications are often multiparty systems that integrate third-party services, e.g., single sign-on (SSO), payment, social networking, from different companies. These applications have significant security requirements. For example, a merchant website that integrates Amazon Simple Pay must ensure that an item is shipped only if it has been paid for. Similarly, an application that integrates Facebook SSO must ensure that only a properly authenticated user should be allowed to log into the application. To implement secure multiparty applications, developers typically follow protocol standards provided by organizations such as OAuth Working Group and OpenID Foundation or API specifications provided by companies such as Amazon, PayPal, and Facebook. However, there is ample evidence that deployed multiparty services are rife with security vulnerabilities. An attacker can purchase without paying [25][27], sign into other people's accounts without passwords [4][28], or get unintended authorizations [29]. The Cloud Security Alliance cites these logic flaws in online services as "Insecure Interfaces and APIs", the No.4 cloud computing threat [10].

We believe that the prevalence of security flaws in multiparty online services calls for rigorous engineering supported by formal program verification. However, despite being advocated by researchers for years, program verification is rarely put in actual engineering practice. This paper presents an approach that significantly lowers the hurdles for real developers to apply formal verifications on online services.

*Protocol-independent security goal***.** The first hurdle in verification is to understand *what security goal to verify*. This is hard for developers because protocol documentation and API specifications are, for practical considerations and limitations, often informal, jargon-laden and incomprehensive. It is unclear what exactly each party is supposed to achieve. For example, an Amazon payment protocol was not explicit about whether the payee's identity was ensured by the cashier or the merchant [27]; many Facebook's relying party websites did not know which piece of Facebook data should be obtained to securely authenticate a user [29].

We realize that security is much more intuitive to be defined as a global property (rather than per-party properties) independent of specific protocols. For example, no matter what a payment protocol instructs a merchant and a cashier to do individually, the two parties should jointly ensure an intuitive global property: for any order to be checked out from the merchant, it must have been paid for on the cashier. This property applies to all payment protocols. We will show concretely that such protocol-independent properties can be defined for various scenarios, which guard an implementation against logic flaws in the protocol and developers' misunderstandings of the protocol.

Certification of Symbolic Transaction (CST). Once a developer has the property, the next hurdle is *how to verify* it. Traditional approaches perform verification *offline* (see Figure 1 (upper)) and face two challenges: (1) the adversary and the system platform need to be modeled precisely (over- or under-permissiveness will lead to false positives or false negatives). The modeled adversary should be able to make arbitrary calls to the web methods on all the parties, with all

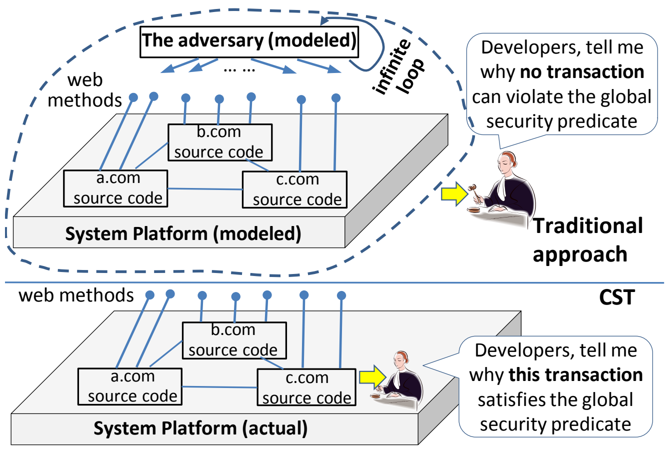


Figure 1: Traditional approach versus CST

possible argument values that the adversary is able to create or obtain. Meanwhile, the system platform should be modeled to constrain the adversary's arbitrariness. The modeling demands considerable insight about the system and the property to be verified. (2) The verifier faces a difficult proof obligation: the logic property must not fail even though the adversary can make an unbounded number (i.e., an infinite loop) of arbitrary calls with arbitrary argument values. The proof requires inductive reasoning about an unbounded number of possible executions. In general, a developer may need to help the verifier by supplying loop invariants and auxiliary lemmas to establish a proof. This is especially hard for real-world source code.

Our technique is called *Certification of Symbolic Transaction (CST)*. It achieves the same verification goal in a different a manner, shown in Figure 1 (lower). It treats every protocol execution (referred to as a *multiparty transaction* from now on) as *a runtime process for creating a proof obligation* for a static program verifier, which we call *the certifier*. The certifier logically examines whether the sequence of computations on all parties *in this transaction* collectively ensures the global predicate. Compared to fully static approaches, the CST approach demands much less from developers: (1) No modeling is needed for the adversary or the system platform, because every transaction to be verified is produced at runtime by a real user interacting with the actual system. (2) Since transactions that fail to be certified are rejected, developers are only required to make sure that every intended transaction is logically sound. This requires the developer to think only about the expected rather than the unexpected scenarios.

An interesting novelty of the CST approach is the use of static program verification at runtime, i.e., to symbolically verify every transaction on-the-fly. In Section II, we motivate why static verification is essential for certifying multiparty transactions. Although program verification is prohibitively expensive in general, CST combines static verification and caching to ensure near-zero amortized overhead. To the best of our knowledge, CST is the first system to use symbolic verification during the execution of a deployed program.

Real-world demonstration. We demonstrate that it is practical to apply CST to real-world systems. We have applied CST on five commercially deployed systems by adding only tens (or 100+) lines of code per party. These systems are based on real-world frameworks, such as PayPal and Amazon Payments services, Microsoft LiveID single-sign-on SDK, the OpenID framework in DotNetOpenAuth, the OAuth template in Visual Studio ASP.NET MVC 4, and NopCommerce software. We also challenged the CST approach by implementing a gambling system integrating four services, for which there is no existing protocol to follow. All the systems and their source code are accessible at https://sites.google.com/site/symbolictransaction/, which is our project page. We have evaluated CST along four dimensions:

- *Security*. We analyzed 14 cases of real vulnerabilities reported in the literature. CST will foil the exploits or avoid the vulnerabilities in 12 out of the 14 cases, the remaining two being out-of-scope issues.
- *Protocol-independence*. We show that different protocols, e.g., Amazon Simple Pay vs. PayPal Standard for payment and OpenID vs. OAuth vs. Live Connect for single-sign-on, can be held to the same global predicates. More interestingly, we show several implementations that blatantly violate the OAuth 2.0 protocol, but satisfy the end-to-end global predicate for single-sign-on. They all turn out to be as secure as the protocol-conformant ones.
- *Performance*. Because symbolic transactions are cacheable, CST incurs only near-zero amortized runtime overhead. Therefore, CST is suitable for real-world deployed services.
- *Programming effort*. We report the lines of code that we added or changed for every open-source package.

II. Overview of CST

The CST approach is a unique combination of concrete runtime execution with symbolic program verification: it verifies each multiparty transaction at runtime but attempts to verify it not just on the actual inputs seen in that transaction but for all possible inputs. To understand this aspect of the design of CST, we first need to recognize two basic characteristics about typical multiparty services:

(1) *There is no global data storage*. Different parties do not fully trust each other despite their cooperation on specific transactions. Consequently, each party holds its own data structures (e.g., databases) locally without a globally-shared storage. For example, PayPal's payment record database is not exposed to any merchant website, because it contains transactions related to other merchants. Similarly, the database of Facebook's single-sign-on service is not

publicly shared with every relying party, because it contains many secret strings and IDs of users of other relying parties.

(2) *Security is a global property*. As discussed in the introduction, security is a property across different parties. For example, "secure checkout" requires that an order on the merchant website has a corresponding payment record on the PayPal server. For this reason, the safety property to be examined by CST needs to refer to data structures of different parties. We call it an *ambient predicate*.

Figure 2 shows a simplified system that preserves the essence of a realistic system. Data structures A[], B and C are defined in the source code of three different companies, respectively. A multiparty application attempts to maintain the ambient predicate (C == true) iff (∃i. A[i] == B) that refers to data structures at every party. Obviously, the ambient predicate cannot be concretely checked, because A[] and B are not shared with *c.com*. An alternative approach and the core idea of CST is to check it symbolically, i.e., to examine whether the sequence of computations of a multiparty transaction *logically* implies the ambient predicate.

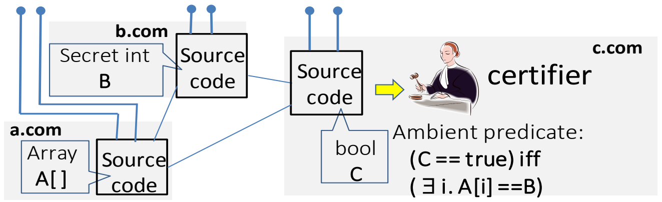


Figure 2: A simple multiparty system.

CST achieves this symbolic verification using a message field called *SymT* (i.e., Symbolic Transaction), attached to every message to collect the source code executed on each party. In general, disclosing source code to another party may lead to intellectual property infringement or reverse-engineering. For CST, this is not a serious concern, because the disclosed source code only consists of a few methods that implements the protocol that every party has agreed on.

When a transaction completes, the certifier uses the final SymT value to synthesize a program representing the executed source code of the entire multiparty transaction. Symbolic verification then checks that the program satisfies the ambient predicate. The collection of the source code and the synthesis of the program to be verified is elaborated in Sections III and IV.

The complexity of verifying a symbolic transaction depends on the expressiveness of the ambient predicate and the program fragments executed by the different parties. We found first-order logic to be adequate for the services we studied and use an off-the-shelf automated program verifier based on satisfiability-modulo-theories [22] for the verification.

In general, symbolic verification is expensive. However, for CST, it incurs an extremely low amortized cost, even lower than what a concrete checking would incur (which would need network messages). The certifier achieves this efficiency by caching the theorem proved by it about a symbolic transaction. Since the theorem holds for all inputs, a future identical symbolic transaction is deemed convincing immediately regardless of the data values on which it computes. If the source code is unchanged, this caching results in near-zero amortized runtime overhead. Furthermore, the caching is over all transactions generated by all users. Most likely, developers themselves are the users who trigger the verification, and real users enjoy the caching.

Threat model. The threat model of CST is the *web attacker model* as defined by Akhawe et al. [2]: the attacker has a browser and his own servers, but does not control the servers of non-attacker parties. Regarding CST specifically, the web attacker model implies that developers of non-attacker parties are cooperative with each other, and do not lie about the executed source code. The network traffic in CST is protected by HTTPS, so the attacker cannot read or tamper with data in transit.

Informally speaking, CST focuses on "logic flaws" in a protocol and its implementation. More formally, these are flaws in the protocol's *symbolic model* as defined by Blanchet [8]: all cryptographic primitives, such as signing and hashing, are considered as black-boxes and assumed to be secure. Also, general programming bugs such as buffer overrun, cross-site scripting and cross-site request forgery, are orthogonal to the type of logic flaws that CST targets. Many techniques have been proposed and deployed to address these issues; these techniques can be used in conjunction with CST.

III. An Example Illustrating CST

We now give an example about secure checkout to explain the CST approach. We first define the ambient predicate for the secure checkout problem. Next, we illustrate a real-world vulnerability that, when exploited, violates the ambient predicate. Finally, we show how CST would have caught the error at runtime.

The basic steps in every checkout transaction are as follows: (1) place an order on the merchant site; (2) make a payment on the cashier site; (3) complete the order on the merchant site. We refer to the sequence of these steps at runtime as a multiparty transaction.

Figure 3 shows the three parties in the transaction – the client and two servers. We assume that *Cashier.com* is the cashier site, and *TStore.com* is the merchant site. The client is a greedy shopper who wants to check out without making a full payment or any payment at all. As the adversary, the client's behavior is arbitrary. Essentially, the "client" can be thought of as the wild Internet that can send arbitrary HTTP requests in any arbitrary order, even those not conforming to Figure 3. A secure implementation must guard against such a malicious client.

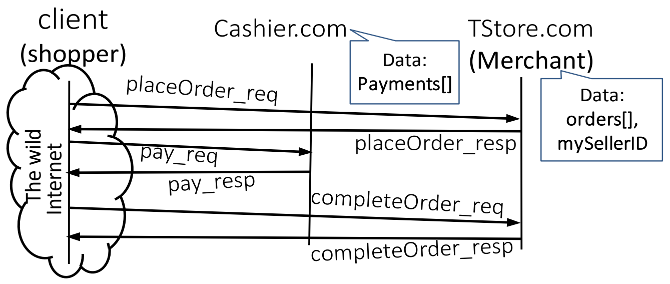


Figure 3: The basic messages and data for checkout.

There are three web methods, placeOrder, pay, and completeOrder. Each is invoked by an HTTP request, and returns an HTTP response. The request and response for placeOrder are denoted placeOrder_req and placeOrder_resp, respectively. Request and response for other methods are named similarly.

Data structures. Every transaction involves the data structures on the two servers: orders[] is an array to store all orders, indexed by the identifier of each order (i.e., orderID); mySellerID is the merchant's identifier registered on the cashier; payments[] is the payment records on the cashier. A real implementation may use database tables instead of arrays. Section V.C will explain how we convert database operations into array accesses by defining "stub methods".

Ambient predicate. An ambient predicate is defined over a fixed multiparty transaction and refers to the fields of request and response of invoked methods and the data structures on the servers. The predicate given below defines secure checkout (the line numbers added for easy reference).

```
placeOrder_req.orderID==completeOrder_req.orderID &&      (1)
∃i. (                                                       (2)
  Cashier.payments[i].status == "Paid" &&                   (3)
  Cashier.payments[i].total
      ==TStore.orders[placeOrder_req.orderID].gross &&      (4)
  Cashier.payments[i].payee == TStore.mySellerID &&         (5)
  Cashier.payments[i].orderID==placeOrder_req.orderID)      (6)
```

The predicate holds for a particular transaction iff there is a payment record at the cashier for the item being bought. Note that this predicate is stated with respect to our problem definition, not specific to any protocol for establishing it.

A. A traditional implementation

The predicate above specifies the security objective. However, it is not locally checkable because it is about data relations across different parties. For example, payments[] is the cashier's data structure, while orders[] is the merchant's. Therefore, it is an ambient predicate.

Protocol specifications today do not explicitly define their ambient predicates. Instead, a protocol simply instructs each party how to check a set of locally checkable predicates and respond to other parties. It is *hoped* that the global safety property is achieved as a result of all these local checks. As explained in the introduction, this is problematic in reality.

LISTING 1: A TRADITIONAL IMPLEMENTATION.

```
class Merchant {
  order_record_t[] orders;
  string mySellerID = "JohnSmith1234";
  PlaceOrderResp_PayReq  placeOrder (PlaceOrderReq req)  {
     PlaceOrderResp_PayReq resp;
     int orderID = req.orderID;   resp.orderID = orderID;
     orders[orderID].status = "Pending";
     resp.redirectionURL = "https://Cashier.com/pay.aspx";
     resp.total = orders[orderID].gross;
     resp.returnURL = "https://TStore.com/completeOrder.aspx";
     sign(resp); return resp;
  }
  public bool completeOrder(PayResp_CompleteOrderReq req){
     if (VerifySignature(req)==false) return null;
     if (req.signer != "Cashier.com" || req.status != "Paid" ||
            orders[req.orderID].status != "Pending") return false;
L1:  orders[req.orderID].status = "Complete"; return true;  }
}
class Cashier {
  payment_record_t[] payments;
  PayResp_CompleteOrderReq pay(PlaceOrderResp_PayReq req){
     if (VerifySignature(req)==false) return null;
     i=getAvailableIndex();
     payments[i].payee = req.signer;
     payments[i].orderID = req.orderID;
     payments[i].total = req.total;
     PayResp_CompleteOrderReq resp;
     resp.redirectionURL = req.returnURL;
     resp.orderID = req.orderID;  resp.status = "Paid";
     sign(resp);  return resp;  }
}
```

Listing 1 shows a simplified example of a traditional implementation. It defines the data structures explained earlier, and implements placeOrder(), completeOrder() and pay() to handle *https://TStore.com/placeOrder.aspx*, *https://TStore.com/compleOrder.aspx* and *https://Cashier.com/pay.aspx*. Let's assume the client checks out a $35 order with orderID 123. In a non-malicious scenario, the messages are as follows (readers can walk through Listing 1 to see how the messages are generated). For brevity, every message is represented by enclosing data fields in angle brackets after the message name, e.g., the first message stands for *https://TStore.com/placeOrder.aspx?orderID=123*.

(1) placeOrder_req:
placeOrder_req<orderID=123>

(2) placeOrder_resp and pay_req (a browser redirection):
pay_req<orderID=123,total=35,returnURL=https://TStore.com/completeOrder.aspx,signature=[TStore's signature for the whole request]>

(3) pay_resp and completeOrder_req (a browser redirection):
completeOrder_req<orderID=123,status=Paid,signature=[Cashier's signature for the whole request]>

Assuming that signing and signature checking are done correctly, readers can confirm that the message sequence above can drive the code to Line L1, where the order is marked *Complete*. However, there is a problem: *the ambient* *predicate we care about is nowhere to be found in* Listing 1. The developers' hope is that the local checks in these methods have collectively ensured "security". Is it really so?

A real-world vulnerability. In fact, this example is based on the real Amazon Simple Pay payment method. An exploitable logic flaw was detailed in Section III.A.2 of reference [27]. In the exploit, the attacker has his own seller account *Mark* and server *MarkStore.com*, and is able to purchase from the victim *TStore* by only paying to *MarkStore*. Specifically, when he receives placeOrder_resp from *TStore*, he discards the signature and re-signs it as *MarkStore*. This message is sent to the cashier (Amazon) as pay_req. From the cashier's point of view, it would seem as if the attacker was purchasing an order from *MarkStore*, so Mark gets paid. However, the redirectURL points to *TStore.com*, so TStore subsequently receives the completeOrder_req signed by the cashier. TStore does not expect the cashier to notify it about an irrelevant payment (i.e., a payment made to Mark), and is fooled to complete the order. NopCommerce [23], a popular e-commerce software, is subject to this flaw.

B. The CST-enhanced implementation

CST enhances the implementation by requiring a SymT field in each message, which contains SHA-1 hash values of the source code of invoked methods. For example, the source code hash of placeOrder is f8f8bd5b0fe4711a09731 f08c06c3749d240580c. For *readability of this paper*, we show a hash value as a hash symbol "#" with a method name, e.g., #placeOrder, but *a real SymT does not contain "#" or method names*.

SymT is now attached to every message shown earlier (parentheses and colons to be explained in Section IV.A, and ε to represent an empty string):

(1) placeOrder_req:
placeOrder_req<orderID=123,**SymT=ε**>

(2) placeOrder_resp and pay_req (a browser redirection):
pay_req<orderID=123,total=35,returnURL=https://TStore.com/completeOrder.aspx,**SymT=TStore.com::#placeOrder()**, signature=[TStore's signature for the whole request, **including the SymT field**]>

(3) pay_resp and completeOrder_req (a browser redirection):
completeOrder_req<orderID=123,status=Paid, **SymT=Cashier.com::#pay(TStore.com::#placeOrder())**, signature=[Cashier's signature for the whole request, **including the SymT field**]>

It is important to note that, for a signed message, such as (2) and (3), the SymT field is covered by the signature, so that the attacker cannot tamper with it. More details about signature checking will be given in Section IV.A.

When (3) arrives at *TStore.com* (the merchant), the last step of computation, completeOrder(...), is invoked (see Figure 4). Thus, the *final SymT* of the transaction is "TStore.com: #completeOrder(Cashier.com::#pay(TStore.com::#placeOrder()))". This SymT string is given to the certifier (which runs

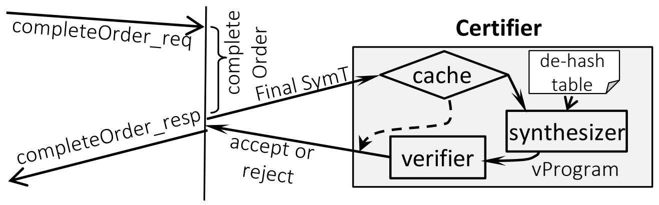


Figure 4: The certifier (on TStore.com).

on *TStore.com*). For now, let's ignore the cache in the figure. When the synthesizer gets the final SymT, it synthesizes a program, namely *vProgram*, which is formally verified against the ambient predicate. If the verification succeeds, the transaction is approved (e.g., the order is marked *Complete*). Otherwise, it is rejected. The synthesis of the program requires the certifier to recover source code texts from their hash values. This capability relies on a "de-hash" table, containing hash-to-source-code mappings. Anyone can submit a source code text to the de-hash table so that its hash value will be computed and associated with it.

A preview of security. The synthesized vProgram for the SymT from the previous paragraph, "TStore.com: #completeOrder(Cashier.com::#pay(TStore.com::#placeOrder()))", has the following steps. First, it invokes the method placeOrder with an arbitrary input value. Next, it invokes the method pay with the return value of placeOrder as the argument. Finally, it invokes the method completeOrder with the return value of placeOrder as the argument. Referring to Listing 1, the reader can see that the ambient predicate is satisfied at the end of this sequence of execution.

On the other hand, if the logic flaw explained in Section III.A is exploited, the final SymT will be TStore.com: #completeOrder(Cashier.com::#pay(MarkStore.com::#placeOrder())). The difference between this SymT and that for the correct transaction is only that the call to method placeOrder is performed at *MarkStore.com,* which, from the perspective of *TStore.com*, has no reason to be trusted. In order words, this step of computation must be considered non-deterministic when *TStore.com* tries to verify the ambient predicate. More specifically, what the synthesizer does is to ignore this step, and begin directly with the method pay with an arbitrary value as argument. Consequently, the certification fails (because merely executing pay and completeOrder on an arbitrary input is not sufficient to satisfy the ambient predicate) and the transaction is rejected. Note that *MarkStore* can even hide its presence by providing SymT=ε in placeOrder_resp. If so, the final SymT will be TStore.com:#completeOrder(Cashier.com::#pay()). All our discussion is still valid in this case.

IV. THE CST CERTIFIER

In this section, we describe the design and implementation of the certifier that validates a symbolic transaction in the CST approach. In particular, we elaborate the synthesizer, the verifier and the cache. Overall, the certifier is a method with three arguments and a Boolean return value:

```
bool certify ( string FinalSymT,
               string AmbientPredicate,
               string[] TrustedParties  )
```

The arguments FinalSymT and AmbientPredicate are self-explanatory. The argument TrustedParties is an array to specify which parties are considered trusted for this ambient predicate. For the example discussed earlier, the certifier (on *TStore.com*) only needs to trust *TStore.com* and *Cashier.com*, i.e., the validity of the ambient predicate should not depend on any other party. Similarly, in the single-sign-on scenario, the certifier on the relying party *foo.com* should only trust *foo.com* and the identity provider (e.g., *facebook.com*), but no one else. As stated earlier, *the client (browser) is always an untrusted party, involved in all transactions*. TrustedParties decides which computation steps the certifier should take into account. Computations performed on other parties, including the client, are ignored in the synthesized vProgram.

A. *Symbolic Transaction*

The symbolic transaction, SymT, is the basis of the CST approach. The SymT representation makes multiparty transactions, hitherto only an informal notion in the mind of a protocol designer, explicit in protocol messages.

SymT needs to record not only the sequence of method calls, but also how two consecutive calls are stitched, i.e., how the output of a call (referred to as method1 on *a.com*) is fed into the input of the next call (referred to as method2 on *b.com*). Specifically, the main question is why *b.com* should believe that the input of method2 indeed comes from *a.com*. There are only two possible reasons: (1) the input is signed by *a.com*; (2) *b.com* itself makes a direct server-to-server call to method1 to obtain the input for method2.

Therefore, SymT must precisely encode the stitching scenarios. Figure 5 shows three SymT values, in which we highlight certain symbols for discussion. In scenario A, the output of method1 is not signed (denoted by the highlighted single-colon), and is supplied to method2 by an unnamed party (denoted by the highlighted parentheses). An unsigned browser redirection is an example of scenario A. The only difference in scenario B is that the input of method2 (i.e., the output of method1) is protected by *a.com*'s signature, so *b.com* is confident that it is generated by *a.com*, untampered. The signing is denoted by the double-colon "::". In scenario C, method1 is called from *b.com* using a direct server-to-server call (e.g., a SOAP or REST API call), so *b.com* of course has the confidence that the input of method2 comes from *a.com*. The server-to-server call is denoted by two pairs of double-parentheses. Another valid SymT, not shown in Figure 5, could be b.com:#method2((a.com::#method1((...)))), representing a server-to-server call that returns a signed response. It is equivalent to scenario C, as the signing is unnecessary.

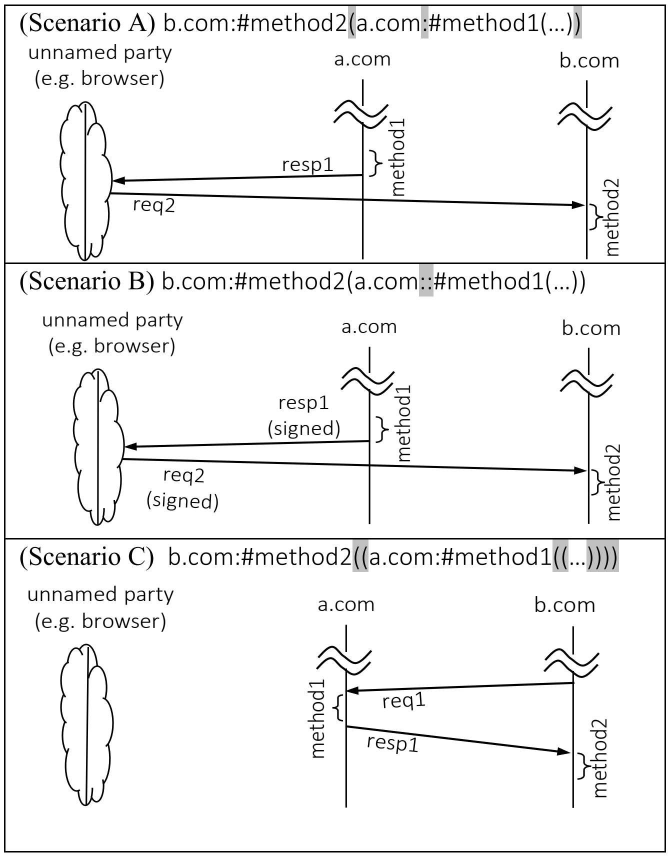

(Scenario A) b.com:#method2(a.com:#method1(...))
unnamed party (e.g. browser)
a.com
b.com
resp1
req2
method1
method2
(Scenario B) b.com:#method2(a.com::#method1(...))
unnamed party (e.g. browser)
a.com
b.com
resp1 (signed)
req2 (signed)
method1
method2
(Scenario C) b.com:#method2((a.com:#method1((...))))
unnamed party (e.g. browser)
a.com
b.com
req1
resp1
method1
method2
Legend: *Double-colon* – signed message
Double parenthesis – server-to-server call

Figure 5: Stitching two method calls.

Signature checking. When receiving a signed message, such as each of the latter two messages in the protocol discussed in Section III, the receiver must check the validity of the signature and whether it matches the SymT string. More specifically, the receiver needs to ensure that: (1) the SymT field itself is covered by the signature; (2) if the SymT is a.com::#method1(...), then the signature is indeed generated by *a.com*.

Signing in CST is simple and can be piggybacked on the existing protocol implementation: if a message in the existing implementation is unsigned, then it remains unsigned in the CST-enhanced implementation; if a message is signed, CST attaches the SymT string as an argument of the HTTP request (GET or POST) and hands the entire message to the underlying signing layer, which signs and sends the request.

Grammar. Grammatically, SymT is a nested sequence of method calls specified by the production rules in Listing 2. The rules we use in our implementation are slightly more comprehensive; they also accommodate a method call taking multiple arguments and a signature covering selective fields.

LISTING 2: PRODUCTION RULES FOR SYMT

```
SymT          → ε
                | PARTY-ID : METHOD-CALL
                | PARTY-ID :: METHOD-CALL
METHOD-CALL → SRC-HASH ( SymT )
                | SRC-HASH (( SymT ))
PARTY-ID      → a.com | b.com | amazon.com | …
```

Semantics. When SymT a.com:f_a(b.com:f_b(c.com:…)) is attached to message M, it represents the following recursive claim about the message (we add brackets to indicate the scoping):

```
The message is M, which a.com claims is
{ the result of executing fa() on input Ma, which b.com claims is
   { the result of executing fb() on input Mb, which c.com claims is
        { … }
   }
}
```

If the above SymT contains double-colons and double-parentheses, the meaning of the claim will not be changed. However, they will affect how the synthesizer trusts each layer of the claim, which will be explained in Section IV.B.

It is worth emphasizing that SymT *must not* be interpreted as follows:

```
a.com claims that {  the message is M,
     which is the result of a.com executing fa() on input Ma,
          which is the result of b.com executing fb() on input Mb,
               which is the result of c.com executing …  }
```

This interpretation is wrong because a.com is not able to make a claim about the whole sequence of calls. The correct interpretation is a recursive claim, in which each party only makes a claim about one call.

B. *Synthesizer*

Think of SymT as an onion potentially rotten inside – each layer is a claim, which, if untrusted, implies that everything inside is bogus. Thus, the synthesizer needs to identify the outer-most layer where the trust cannot be established, and discard it with everything inside. Specifically, the synthesizer examines the SymT string from left to right. It looks for the first call which is:

(1) performed at an untrusted party (i.e., PARTY-ID $\notin$ TrustedParties), or

(2) not tamper-proof (i.e., when the pattern "(PARTY-ID:METHOD-CALL)" is matched, such as in Scenario A).

If such a call is found, it is discarded and replaced with the empty string ε. The resulting SymT, basically a hollow onion, is trusted. The vProgram can be directly generated from it without any further considerations regarding trust. It takes arbitrary input values.

Listing 3 shows the synthesized program corresponding to the SymT TStore.com:#completeOrder(Cashier.com::#pay(TStore.com::#placeOrder())). The method to be verified is the static method main(). The local variables of this method, such as placeOrder_req, pay_req, etc., and the global

LISTING 3: THE VPROGRAM SYNTHESIZED FROM SYMT.

```
class vProgram {
  /* The constructors of Merchant and Cashier construct objects
   with non-deterministic initial states */
   static Merchant TStore=new Merchant();
   static Cashier MyCashier=new Cashier();
   static void main() {
   /* The program for a normal transaction will contain L1 and L2.
   The program for the attack described in Section III.A will not
   contain L1 and L2. */
L1: placeOrder_resp=TStore.placeOrder(placeOrder_req);
L2: pay_req = placeOrder_resp;
L3: pay_resp = MyCashier.pay(pay_req);
L4: completeOrder_req = pay_resp;
L5: bool completeOrder_resp=
              TStore.completeOrder(completeOrder_req);
L6: if (!completeOrder_resp) return;
L7: Contract.Assert(placeOrder_req.orderID==
                    completeOrder_req.orderID);
L8: Contract.Assert(
     Contract.Exists(0,MyCashier.payments.Length, i =>
       MyCashier.payments[i].status == "Paid" &&
       MyCashier.payments[i].total ==
          TStore.orders[completeOrder_req.orderID].gross &&
       MyCashier.payments[i].payee ==
                     TStore.mySellerID     &&
       MyCashier.payments[i].orderID==
                     completeOrder_req.orderID )
     );
   }
}
```

objects, such as TStore and MyCashier, are initialized with non-deterministic values; this initialization is not shown in the figure. Since all method calls happen at trusted parties and no method call is an instance of Scenario A, lines L1-L6 compose all the method calls. Lines L7 and L8 are reached only if the order is completed. These lines assert that the ambient predicate holds on the preceding computation. On the other hand, the SymT for the attack from Section III.A is "TStore.com:#completeOrder(Cashier.com::#pay(MarkStore.com::#placeOrder()))". In this SymT, the computation of MarkStore.com::#placeOrder() is at an untrusted party and is therefore replaced by ε, causing lines L1 and L2 to be dropped, so the assertions will fail. Similarly, if the first step was (TStore.com:#placeOrder()), which is not tamper-proof, it would result in the same vProgram without lines L1 and L2. Appendix B illustrates the synthesis steps graphically.

In addition to the method main(), the complete vProgram also includes the aforementioned constructors and web methods, which are hashed and stored in the certifier's de-hash table.

C. *Verifier*

We demonstrated CST on systems implemented using ASP.NET and C#. The focus on .NET is only because we want to use an off-the-shelf program verifier for .NET. The

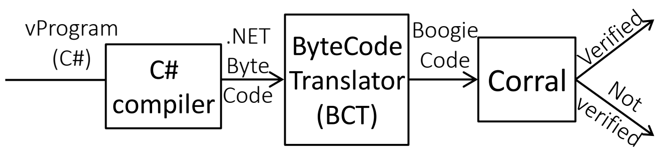


Figure 6: The verifier.

CST technique is equally applicable to any programming language. Figure 6 shows the tool chain. The program generated by the synthesis tool is compiled by the C# compiler of Visual Studio. The output is an executable file of .NET byte code. ByteCodeTranslator (BCT) [5] is a tool to translate a .NET byte code program into a Boogie program. Boogie is an intermediate verification language [9]. We use the Corral system [16] as the verifier. In addition to the input Boogie program, the Corral verifier expects a non-negative number to establish a bound for the unfolding of loops and recursion in the program. Corral outputs exactly one of three results: the program is verified, or the program is verified with respect to the bound, or the program is falsified. In the final case, Corral also presents a counterexample witnessing the error in the program. Our certifier certifies a transaction *only if Corral returns the first output, i.e., the transaction is verified without a bound.*

Ambient predicates, like lines L7 and L8 in Listing 3, are specified using the `Contract` class [20] defined in C# `System.Diagnostics.Contracts` namespace.

D. Cache

Program verification is expensive (e.g., 10 - 30 seconds to verify a transaction in our cases). It is impractical to do it for every transaction. Caching is therefore essential in CST: when the verification is done, the certifier caches the result (i.e., true/false) with the triple `FinalSymT`, `AmbientPredicate` and `TrustedParties`. Any future call to the certifier by any user will return the result directly if it matches a cached triple.

E. Putting it all together

A one-sentence summary of CST's safety guarantee is that *each certified transaction is assured to satisfy the ambient predicate regardless of the value of the current state of the multiparty system*. This assurance comes from three elements in the CST design, which fulfill different but complementary goals:

(1) *SymT construction*. The SymT construction is to passively record the computations. It faithfully states important basic facts, such as whether a message is a redirection or direct server-to-server call, and whether a message is signed (if so, who signs it).

(2) *vProgram synthesis*. The synthesis is to recover the computations based on the SymT string. As explained earlier, the SymT string can be thought of as an onion. The untrusted inner core is discarded, so the remaining hollow onion represents the truth about the computations in the transaction.

(3) *vProgram verification*. It symbolically checks whether the sequence of recovered computations is sufficient to ensure the ambient predicate. This is done by an off-the-shelf verification tool chain.

F. Important practical considerations

Ambient predicate and replay attack. The three elements above only ensure that the transaction satisfies the ambient predicate. Whether this is sufficient to ensure security depends on the definition of the ambient predicate. As an example, we consider the possibility of replay attacks on the example discussed in Section III. If the clause `Cashier.payments[i].orderID==placeOrder_req.orderID` was missing from the ambient predicate, CST would not guard against a replay attack that makes only one payment but checks out multiple orders with the identical total price from the same merchant. Obviously, this attack will have serious consequences.

Typically, a protocol designed to be resilient to replay attacks contains some sort of sequence number (or nonce). To prove that resilience to replay attacks is indeed achieved, the ambient predicate should check the sequence number (or nonce) to make sure that each transaction is honored only once, i.e., it is void when replayed. The `orderID` clause above serves this purpose. It ensures that there is a one-one mapping between the payment ID and the order ID, thereby ensuring that the payment is for the order being checked out.

Revoking the effect of a rejected transaction. CST checks the safety property at the last moment before a transaction is completed. Hence, a rejected transaction may have changed the state on the involved parties, e.g., a payment is made but the final checkout step is rejected. Undoing these effects for a rejected transaction requires an explicit revocation procedure.

Even without CST, a protocol may abort an on-going transaction at any step for a variety of reasons. Consequently, a real-world system already requires a mechanism handle such situations, e.g., payment dispute, automatic revocation, etc. A transaction rejected by CST's certifier is not fundamentally different from those rejected by other conditions. The developer does not need to perform any special handling for CST and only needs to add the CST check as a conjunct to the existing rejection conditions.

V. Applying CST in the Real World

We have applied CST to enhance various systems that serve practical purposes. Unlike proof-of-concept prototypes, these systems contain realistic source code and data structures written by actual developers. We view it as an accomplishment that all our enhanced systems are ready for commercial deployments. For example, people can install our CST-enhanced NopCommerce to run their stores: a customer can choose items, check out orders, and specify shipping and payment methods, etc; payments are made on the real Amazon and PayPal servers. People can also use our Live Connect SDK to enable single-sign-on on their websites. Functionality-wise and performance-wise, our systems are indistinguishable from the original ones. These systems are all publicly accessible. Their URLs, source code, as well as instructions and videos of demos, are given in https://sites.google.com/site/symbolictransaction/ [32].

A. Categories of the enhanced systems

We have worked on three categories of systems so far:

Payment/checkout. NopCommerce [23] is a widely used open-source e-commerce application. It was one of the focused systems in previous security studies [27][30]. NopCommerce accepts many third-party payments. We decided to enhance its payment modules for Amazon Simple Pay and PayPal Standard. The former is essentially what we described in Section III.A, and latter is shown in Figure 7. They are significantly different in that Amazon Simple Pay is based on signed redirection messages, whereas the PayPal Standard mechanism relies on a direct server-to-server call, namely the PDT (Payment Data Transfer) query, for securely communicating the payment details.

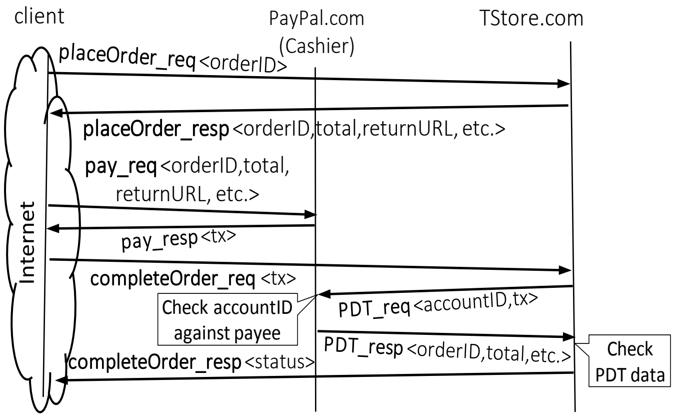

client
PayPal.com (Cashier)
TStore.com
Internet
placeOrder_req<orderID>
placeOrder_resp<orderID,total,returnURL, etc.>
pay_req<orderID,total, returnURL, etc.>
pay_resp<tx>
completeOrder_req <tx>
PDT_req<accountID,tx>
Check accountID against payee
PDT_resp<orderID,total,etc.>
Check PDT data
completeOrder_resp<status>

Figure 7: PayPal Standard.

Single-sign-on (SSO). We worked on the implementations of three different SSO protocols: (i) the *OpenID-2.0*-based SSO [24] in the DotNetOpenAuth framework [12]; (ii) the *OAuth-2.0*-based SSO [15] in Microsoft Visual Studio ASP.NET MVC 4 web application template that uses Facebook's OAuth service [13]; (iii) Live Connect SDK [19], which heavily influenced the *OpenID Connect* specification [26]. (Note that the terminology may cause a little confusion. *OpenID Connect* is a protocol, drafted by the OpenID Foundation, to use OAuth 2.0 for SSO. It was published very recently. Live Connect SDK predates the OpenID Connect specification, so the SDK refers to its SSO mechanism as OAuth 2.0, rather than OpenID Connect.) The message diagram of the OAuth-2.0-based SSO is shown in Figure 8. The ones for OpenID 2.0 and Live Connect are given in Appendix A.

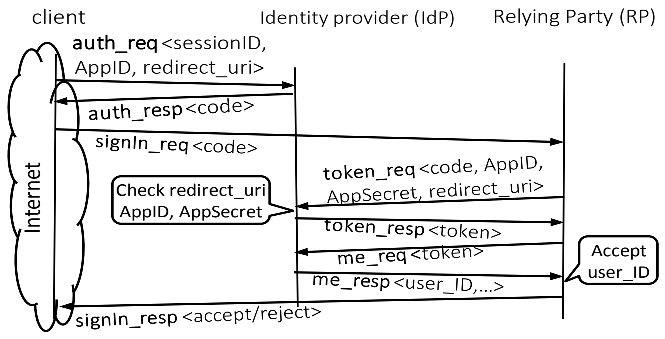


Figure 8: SSO based on OAuth-2.0.

Gambling. People are familiar with the above two categories, because standards organizations and major companies have provided protocol specifications or API documentations. We decided to use CST to build a gambling system. The goal is two-fold: (1) we do not have any existing gambling protocol to conform to, so building this system is an end-to-end exercise of the protocol-independent thinking process; (2) previous scenarios only involve two trusted services. We want to challenge the CST approach by involving more parties. The gambling system we built consists of four independent services for betting, payment, authorization and coin-tossing (see Figure 9).

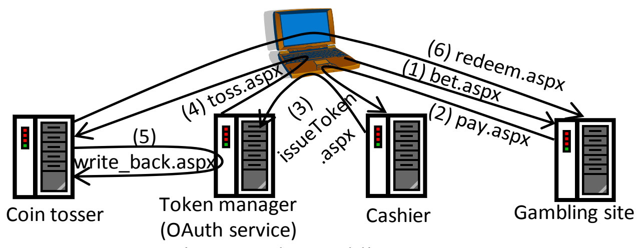


Figure 9: The gambling system.

B. Ambient predicates

Despite the significant differences among these systems and their adopted protocols, we specify the same ambient predicate for each category.

Payment/checkout. Our enhanced implementations for Amazon Simple Pay and PayPal Standard ensure the same ambient predicate as we presented in the example in Section III. It is to ensure that when the merchant is about to check out an order, there exists a payment record in the cashier that matches this order.

SSO. Figure 10 gives our protocol-independent definition of the SSO security goal. In every SSO system, there are a client, an identity provider (IdP) and a relying party (RP). The client holds a piece of BrowserSecret, which is shared with the identity provider, but not the relying party. The relying party has at least two constants: My_Realm is its identifier known to the identity provider; My_Hostname is its network-addressable name. Variable Session_UID is the session variable to be set upon a successful sign-on.

An SSO transaction starts with a request from the client to the identity provider, namely auth_req, containing the

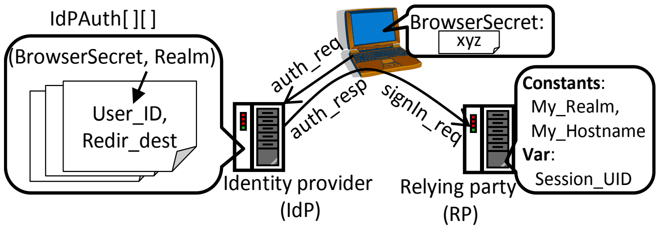


Figure 10: A protocol-independent definition of single-sign-on.

BrowserSecret and the Realm of the relying party that the client wants to sign in. The identity provider then retrieves an object called ID_Claim using the pair (BrowserSecret,Realm). ID_Claim contains at least two fields: User_ID is the identifier of the user that this claim is about; Redir_dest indicates the destination of the redirection message (i.e., the website that signIn_req is sent to). The retrieval is based on a two-key dictionary called IdpAuth, defined as follows in C#:

```
Dictionary<string, Dictionary<string, ID_Claim>>  IdpAuth;
```

Note that how IdpAuth entries are established is not what SSO concerns about. The identity provider can identify the client as "Alice" for any reason (e.g., through password or SSL client certificate), thus creates an IdpAuth entry. An SSO protocol is to prove to the relying party the existence of the entry, i.e., the fact that the identity provider believes the client is Alice.

An SSO transaction must satisfy the ambient predicate:

```
IdPAuth[auth_req.BrowserSecret][My_Realm].Redir_dest
                                 == My_Hostname   &&        (1)
IdPAuth[auth_req.BrowserSecret][My_Realm].User_ID
                                 == Session_UID             (2)
```

The first clause asserts that the identity provider passes the ID_Claim to *this* relying party, not to any other website (which could then use the ID_Claim to sign into *this* relying party illegally). The second clause asserts that the user ID to be associated with the session (i.e., Session_UID) is the one in the aforementioned ID_Claim.

Gambling. The ambient predicate for the gambling system is given below. Clauses (2)-(4) ensures that a proper payment has been made for the bet (identified by final_req.betID); Clauses (5) - (8) ensures that the bet is valid and matches the tossing result of the coin-tosser.

```
GamblingSite.bets[final_req.betID].status=="Pending" &&          (1)
∃ i. (
  Cashier.payments[i].total ==
        GamblingSite.bets[final_req.betID].amount    &&          (2)
  Cashier.payments[i].orderID  ==  final_req.betID   &&          (3)
  Cashier.payments[i].payee==GamblingSite.MySellerID)    &&      (4)
∃ x. (
  TokenMgr.records[x].payee==GamblingSite.MySellerID &&          (5)
  TokenMgr.records[x].betID  ==  final_req.betID   &&            (6)
  TokenMgr.records[x].EffectiveResult  !=  "untossed"  &&        (7)
  GamblingSite.bets[final_req.betID].guess
                 ==TokenMgr.records[x].EffectiveResult)          (8)
```

As mentioned earlier, a motivation for building this gambling system is to challenge the CST approach with substantial complexity. In this case, the final SymT of a normal transaction contains 4 parties and 7 hash values:

```
GamblingSite.com:#redeem(CoinTosser.com::#post_toss((Toke
nMgr.com:#write_back((CoinTosser.com:#toss(TokenMgr.com::
#issueToken(amazon.com::#pay(GamblingSite.com::#bet()))))))))
```

The synthesized vProgram has more than 300 lines of C# code, which the certifier is able to verify.

C. *Programming*

Every verification technology applied to real-world systems needs the effort of abstraction. The abstraction is often done through *factoring* and *stubbing*. This subsection explains what they mean in the CST programming. For concreteness, the description incorporates our experience of enhancing the Live Connect SDK, although the general ideas apply to our experiences of enhancing other systems.

(1) Factoring. CST requires the core computations to be factored out in order to be logically verified. Typical non-core computations include methods for parsing, composing and HTTP-encoding/decoding for messages. These non-core computations contain complicated string (byte-array) operations. Currently Corral and Boogie have only limited capability for reasoning about string operations: assignment, equality comparison and string-indexed dictionary are supported, but concatenation, tokenization, character operations, etc. are not.

Figure 11 shows the call-graph of callback.aspx in the Live Connect SDK, which handles a redirection from the LiveID server. In this 3-dimensional drawing, the methods are placed on an unshaded level and a shaded level. The shaded level consists of the core methods that we factor out. The unshaded level serves as the interface between the core logic and the underlying platform. For example, Page_Load parses HTTP arguments. RequestAccessToken_raw is a method we create so that the string operation for constructing an HTTP request can be separate from the core method RequestAccessToken. The constructor of class JWT (i.e., JsonWebToken) performs Base64 decoding and signature validation, which are byte-array operations.

It is not a requirement that all complicated string (byte-array) operations are moved out from the core methods. For example, a core method can still construct a string for the

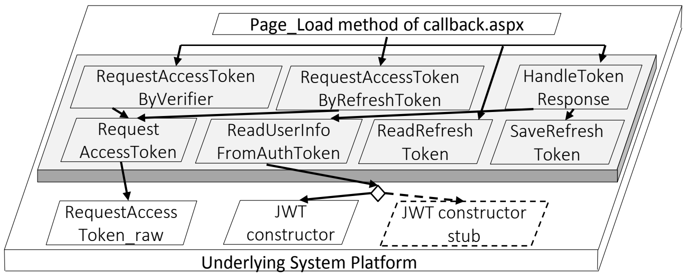

Page_Load method of callback.aspx
RequestAccessToken ByVerifier
RequestAccessToken ByRefreshToken
HandleToken Response
Request AccessToken
ReadUserInfo FromAuthToken
ReadRefresh Token
SaveRefresh Token
RequestAccess Token_raw
JWT constructor
JWT constructor stub
Underlying System Platform

Figure 11: Factoring and stubbing in callback.aspx.

logging/debugging purpose, as long as it does not affect the validity of the ambient predicate. Our experience on existing implementations of real-world service frameworks is that they are already architected similarly to Figure 11 so that lower-level methods parse HTTP requests into well-structured objects and assemble HTTP responses using these objects, while upper-level modules implement core computations on these objects. The core computations usually deal with basic types (e.g., integers and Booleans), structs and arrays of basic types, as well as string assignments and equality comparisons. Corral/Boogie can effectively reason about all these programming constructs.

(2) Stubbing. The core methods call many other methods, which will not be included in the vProgram for verification. In other words, these methods are treated as unimplemented, from the certifier's standpoint. For Corral/Boogie, the default semantics of an unimplemented method is that it returns a non-deterministic value, but does not modify any program state (i.e., the body is a no-op). This works in most cases, because most of these methods are not essential to the verification. However, there are a few situations in which the semantics of these methods matter, so developers need to define their semantics as stubs. In the Live Connect SDK, we provided a stub method as shown in Figure 11. The source code is below.

```
static JsonWebToken JWT_Constructor_stub (OAuthToken token)
{     JsonWebToken jwt;
      havoc(jwt);   //to assign jwt a non-deterministic value
      Contract.Assume(jwt == token.jwt && jwt != null);
      return jwt;
}
```

The reason to provide JWT_Constructor_stub is to replace the Base64 decoding and signature validation operations in the JWT constructor with the logic most essential to the verification. Specifically, the logic is that a new JWT object equals to the jwt member of the input argument token, i.e., jwt!=null && jwt== token.jwt.

Another situation for providing a stub is to model a database operation. In real-world systems, persistent data (e.g., the payment records) are often stored in and queried from a database by INSERT and SELECT. Corral/Boogie does not have built-in support for these operations. Developers need to wrap these operations in C# methods, and define stubs that are logically equivalent to database operations but use C# data structures like array, set, list, etc.

(3) Mapping from variable names in ambient predicate to those in implementation. Every verification technique needs to map variable names in specification to that in source code, and so does CST. The ambient predicates defined above use generic names for message fields and variables. Unfortunately, they are named differently across protocol documentation and implementations. In our programming, we had to adapt the ambient predicates to the terminologies of these implementations. For example, the BrowserSecret in our definition is called MSPAuth in Live Connect; the Realm in our definition is called AppID in Facebook OAuth and openid.realm in OpenID 2.0.

D. *Deployment*

The deployment path we envision is that, first, major service providers, e.g., Facebook and Amazon, attach the SymT field in their messages; then, relying websites gradually opt-in to take advantage of CST. Note that CST has the advantage of incremental deployment: without any modification, a CST-unaware relying website (i.e., a relying party or merchant website) will just work normally with a CST-enhanced service provider, as SymT is treated as a superfluous field.

Even before the service providers actually deploy the enhancement, CST can be used to secure real-world transactions. This is exactly what we did for all the aforementioned open-source packages (except for DotNetOpenAuth), which do not contain the service provider code. For each of these services, we built a "wrapper service", which serves as a relay in order to attach the SymT field. An example is shown in Figure 12, the dashed-line redirection is implemented as two solid-line ones with the wrapper service as the trampoline. A server-to-server call is similarly implemented. Hence, all transactions certified on our relying websites are processed by the actual service providers for real. Of course, a caveat is that the attached source code hash is only our best effort approximation for the logic behind these web methods.

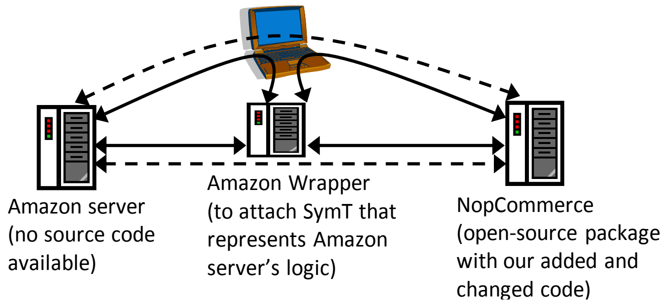


Figure 12: An example wrapper service.

VI. Evaluation

We evaluated the CST approach on security, protocol-independence, performance, and programming effort.

A. *Security*

We studied 14 real-world vulnerabilities (listed in Table I), which we believe are a representative sample set in this problem space – this set includes all the cases reported in the literature [25][27][28][29] that allows an attacker to either check out an order without a proper payment or sign into a victim user's account through SSO, excluding the cases due to generic web programming flaws like cross-site scripting, cross-site request forgery (CSRF) and session fixation. We show next that 12 out of the 14 cases would be addressed by CST. The other two are out-of-scope issues.

Cases for which CST is effective. Attacks for cases #1, #2 and #11 can be launched against the systems that we built using CST – NopCommerce with PayPal Standard, NopCommerce with Amazon Simple Pay and OAuth-2.0-based SSO. We confirmed that the attacks result in vPrograms not satisfying the ambient predicates.

Case #7 is about a relying party that uses the email address (email) field, rather than the claimed_id field, as the user's identifier. The reported vulnerability is because the signIn_req.email field can be excluded from the signature coverage by the malicious user, so that it bears an arbitrary value supplied by the client. We intentionally introduced this vulnerability to our OpenID 2.0 implementation on DotNetOpenAuth. When the attack is launched, the resulting vProgram fails to verify clause (2) in the ambient predicate, because Session_UID is taken from the signIn_req.email field, which is non-deterministic in the vProgram.

Cases #3-#5 are about Interspire, which is another merchant software providing similar functionalities as NopCommerce. We have not applied CST on Interspire. However, based on the nature of the attacks, it is clear that they fall nicely into the scope of CST:

- Case #3 is an attack in which the attacker starts two independent transactions – one is expensive, the other is cheap. The attacker only performs the PayPal payment step in the cheap transaction, but not in the expensive transaction. At a particular stage, the merchant takes a signed orderID as the input argument. It is at this stage where the attacker supplies the signed orderID of the expensive transaction into the HTTP session of the cheap transaction, so the expensive order is checked out although only the cheap order is paid. This attack will be defeated by CST, because the orderID is always attached with the SymT, and signed together. When the attacker swaps the orderID of the expensive transaction into the cheap transaction, the SymT of the transaction has to be swapped in as well. The SymT clearly indicates that no payment step has been performed, so the ambient predicate will fail to verify.
- Case #4 is a vulnerability because the merchant may take the orderID from the client's cookie that is not signed. According to our definition, any unsigned value supplied by the client is non-deterministic. Having a non-deterministic orderID, the vProgram fails verify.
- Case #5 is because the payment total is calculated based on the shopping cart at the checkout time, but the order being checked out is generated based on the shopping cart after the payment is made. The ambient predicate will not verify in this case, because the shopping cart is a runtime object, querying its property at two time points are semantically two method calls, corresponding to two different symbolic values. The equality would not be established in the verification.

Case #9 is about JanRain SSO service. The attack is to set the redirection destination (i.e., Redir_dest) to the attacker's website when the (victim) user tries to sign into a (victim) website. The JanRain server correctly checks the redirection destination, but the most important step in the attack is that the client can swap in an unchecked URL as the redirection destination after the checking. If CST was applied, the unchecked URL would be an arbitrary value (i.e., the attached SymT would not indicate any logic constraint imposed on this URL), so the clause in the ambient predicate about the redirection destination would fail.

Similar to case #7, case #10 is about a relying party that intends to use the email field as user ID. However, the developer mistakenly uses an arbitrary non-email field as the email field, due to a misunderstanding of the OpenID 2.0 protocol. CST would prevent the flaw because the IdPAuth dictionary on the identity provider would not even contain this arbitrary field.

Cases #12-#14 include every exploitable flaw reported in [25]. In case #12, the attacker replaces the payee account ID with his own PayPal account ID, and checks out an order from the victim store by paying himself. Cases #13 and #14 are similar to case #3, in which the attacker places two orders in two sessions, and supplies a message obtained from the session of the cheaper order into the session of the more expensive order. We have explained that these are precisely the type of logic flaws that CST would prevent.

TABLE I: REAL-WORLD CASES STUDIED IN OUR SECURITY ANALYSIS

No.	Attack	Target system	CST effectiveness
#1	Section III.A.1 of [27]	NopCommerce with PayPal Standard	Yes
#2	Section III.A.2 of [27]	NopCommerce with Amazon Simple Pay	Yes
#3	Section III.B.1 of [27]	Interspire with PayPal Express	Yes
#4	Section III.B.2 of [27]	Interspire with PayPal Standard	Yes
#5	Section III.B.3 of [27]	Interspire with Google Checkout	Yes
#6	Section III.C of [27]	Websites using Amazon Payments	No
#7	Section 4.1 of [28]	Websites using Google ID	Yes
#8	Section 4.2 of [28]	Websites using Facebook Connect	No
#9	Section 4.3 of [28]	Websites using JanRain sign-on	Yes
#10	Section 4.5 of [28]	Websites using Google ID	Yes
#11	Section 2 of [29]	Websites using OAuth implicit flow for SSO	Yes
#12	Section IV.A.1 of [25]	osCommerce, CS-Cart and AbanteCart using PayPal Standard	Yes
#13	Section IV.A.2 of [25]	OpenCart and TomatoCart using PayPal Express	Yes
#14	Section IV.A.3 of [25]	TomatoCart using PayPal Express	Yes

Cases that are not addressed by CST. CST relies on every party to correctly verify signatures. Case #6 is a vulnerability in signature verification. It is out of scope of CST. The root cause of case #8 is a client-side cross domain issue. Specifically, it is due to a special Adobe Flash communication mode that does not conform to the same-origin policy. This causes secret data from the IdP to be obtained by a malicious webpage on the victim user's browser. CST does not address security flaws in the underlying platform.

B. *Protocol independence*

The fact that we check the same ambient predicates for systems adopting considerably different protocols shows the protocol independence of their security goals. To make the point even stronger, we built implementations that *blatantly violate protocols but are nevertheless secure*. Three of them are shown in Figure 13.

Implementation (A) does not conform to the OAuth 2.0 protocol (shown earlier in Figure 8) for two reasons:

• *First*, the protocol requires token_req to contain the field AppSecret, which is a secret the identity provider assigned to every relying party at the registration time. We realized that the purpose of AppSecret is to prevent another website from impersonating the relying party to access the identity provider, which seemed unrelated to SSO. We removed AppSecret, and the ambient predicate still held, suggesting that AppSecret is indeed unnecessary in SSO. Note that we do not claim that AppSecret is useless in general in the OAuth protocol (in fact, we now understand precisely where it is useful).

• *Second*, the protocol requires the identity provider to check redirect_uri. In implementation (A), the identity provider does not perform the check. Instead, it returns redirect_uri in me_resp, so that the relying party can check it. The ambient predicate still hold in this case.

Implementation (B) even more blatantly violates the protocol, because it gets rid of code, but uses token for the client to authenticate into RP.com. According to a previous study, using token to authenticate is a pervasive and serious vulnerability [29]. Recently, the OAuth 2.0 specification has been augmented to explicitly forbid this kind of token usage (see section 10.16 of RFC 6749). However, we realized that this usage is vulnerable only because, if the relying party accepts a token rather than a code for authentication, the steps token_req and token_resp will be skipped. The relying party directly calls me_req:(token), so the checking of redirect_uri and AppID required by the ambient predicate is missing. If me_req took additional arguments redirect_uri and AppID, and the identity provider performed the checking, as in implementation (B), then security would still be achieved.

Suppose the identity provider insists not to check anything, is the implementation doomed flawed? Not necessarily. The checking can be performed by the relying party, like in implementation (C).

Summary. We can see that all these implementations are just different ways of sharing the responsibility of performing all necessary checks. The OAuth 2.0 protocol describes one particular way, but not the only way. Of course, we understand that protocol conformance not only affects security, but also modularity, deployability, interoperability, etc. We do not suggest implementers disregard protocols, but only argue that security can (and should) be ensured independently, because understanding "who should do what, and why" about each protocol specification can be very subtle.

C. *Performance*

A significant strength of CST is its near-zero runtime overhead. Table II provides the measurement results, obtained from a server with a 2.10 GHz CPU and a 3.5 GB RAM, running Windows Server 2008. The numbers fall into two categories: per transaction cost and one-time cost. The time spent on synthesizing and verifying a vProgram belongs to the one-time cost, because the caching amortizes

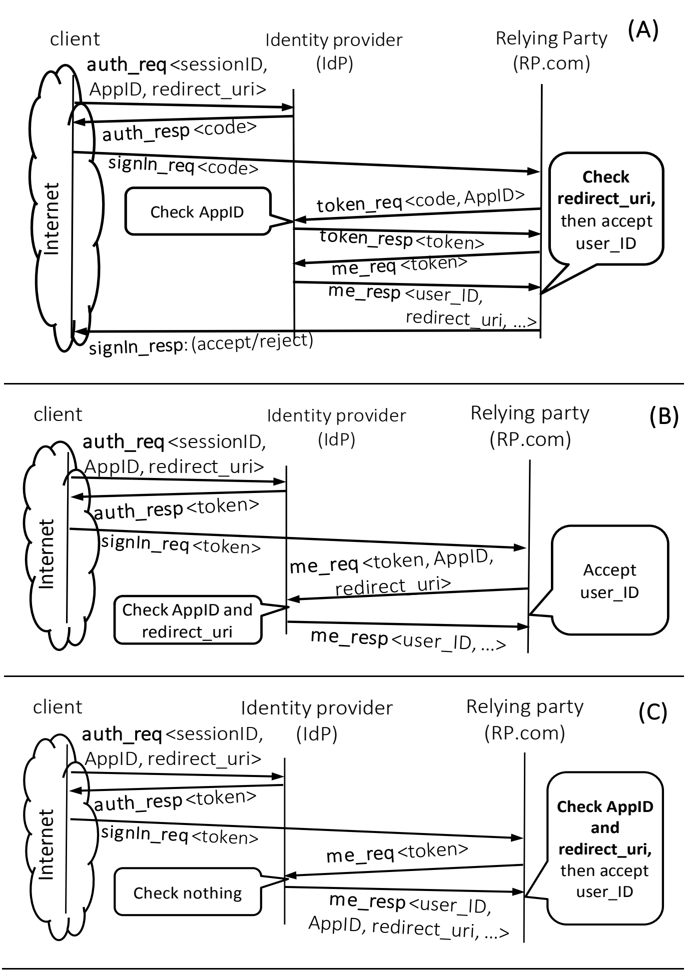


Figure 13: Protocol-violating yet secure implementations.

TABLE II. PERFORMANCE OVERHEAD – PER TRANSACTION AND ONE-TIME COSTS.

	Per-transaction cost		One-time cost		
	Runtime overhead	Average traffic overhead	Program synthesis using a local de-hash server	Program synthesis using a remote de-hash server	compilation, byte-code translation and verification
Live Connect SDK	≅ 0ms	106 B/SymT	3ms	568ms	18758ms
OpenID 2.0 on DotNetOpenAuth	≅ 0ms	119 B/SymT	5ms	409ms	15380ms
Facebook SSO using ASP.NET MVC 4	≅ 0ms	120 B/SymT	5ms	408ms	12090ms
NopCommerce with Amazon Simple Pay	≅ 0ms	78 B/SymT	2ms	450ms	15444ms
NopCommerce with PayPal Standard	≅ 0ms	105 B/SymT	8ms	190ms	10990ms
Coin tossing gambling	≅ 0ms	205 B/SymT	3ms	945ms	32477ms

the cost over all transactions on all users. In fact, developers themselves are most likely the users who actually pay for the cost during testing.

Per-transaction cost. For a non-certifier party, the only runtime overhead is to produce the SymT. The source code hash is a pre-computed constant for a given version, so the only overhead is a string concatenation. Also note that *CST incurs no additional signing operation*, i.e., any unsigned message in the original implementation will remain unsigned in the CST-enhanced implementation. For the certifier, the only per-transaction overhead is the cache lookup for the SymT. Obviously, the runtime overheads for both a non-certifier party and the certifier should be extremely small. We nevertheless did the actual measurements to confirm that, for every system we implemented, the per-transaction runtime overhead is too small to report.

The SymT field incurs traffic overhead for protocol messages. We measured the average traffic overhead per SymT field (shown as Bytes/SymT). Our implementations use SHA-1 (160 bits), RSA (384 bits) and UTF-8 for hashing, encryption and encoding.

One-time cost. The synthesis cost is measured for two situations – when the de-hash table is stored locally or on another server. The first one mainly indicates the computational time of the synthesis algorithm, which is within 5 milliseconds in each of our case. The second situation may be more beneficial in practice because it offloads the de-hash table to another server. Although the synthesis time is longer, since it is a one-time cost, it should not be a performance concern in practice.

The last column in Table II corresponds to the real heavy-lifting step in CST. It consists of C# compilation into .NET byte code, byte-code translation into Boogie code and verification of Boogie code. The time reflects the significant logic complexity for verifying a transaction consisting of realistic methods. In contrast, today, this significant logic reasoning is never conducted, and correctness is taken on faith.

D. Programming effort

Table III shows the lines of code (LoC) we added or changed in each open-source project, excluding comment and white lines. The certifier is the same across all projects. It consists of 347 LoC. The LoC numbers in the unshaded cells are a good measurement of the effort for factoring and stubbing. The amount of code is fairly small, under 200 LoC for each party, indicating that the original developers had architected the code well so that it was amenable for the CST enhancement. The shaded cells correspond to our wrapper code for the real API providers, and factoring and stubbing do not apply for them.

TABLE III: LINES OF CODE THAT WE ADDED OR CHANGED IN THE OPEN-SOURCE PACKAGES (COMMENT AND WHITE LINES EXCLUDED)

	Shared methods	The relying website	The API-provider
Live Connect SDK	0	48	100 (wrapper)
OpenID 2.0 on DotNetOpenAuth	104	59	182
Facebook SSO using ASP.NET MVC 4	0	119	411 (wrapper)
NopCommerce with Amazon Simple Pay	0	71	375 (wrapper)
NopCommerce with PayPal Standard	0	71	239 (wrapper)

VII. RELATED WORK

There is a rich body of literature about verifying security protocols themselves, which we do not discuss here due to the space constraint. Research is also conducted to address issues in protocol implementations. Existing approaches can be categorized as either *top-down* or *bottom-up*. The top-down approaches focus on generating or verifying implementations based on formal specifications of protocols. For example, Bhargavan et al. [6] verified a number of reference implementations of the InfoCard protocol. In their work, the protocol and the security specifications are written in high-level languages F# and WSDL. Bhargavan and Corin et al. [7][11] developed a compiler that can synthesize a protocol implementation from a high-level F# specification of multiparty transactions. The bottom-up approaches try to extract protocols from actual systems. Aizatulin et al. [1] proposed to use symbolic execution to convert a protocol implementation in C into its high-level model in the applied pi calculus. Bai et al. developed a technique to extract SSO protocols from HTTP messages of network traces [4]. The uniqueness of CST is that it performs static verification at runtime, which converts the harder obligation of verifying a system into that of verifying intended transactions.

Proof carrying code (PCC) [21] is a technology for a code consumer (e.g., an OS kernel) to examine whether the code from an untrusted producer (e.g., a kernel extension from a third-party company) is accompanied by a logic proof of desired safety properties. CST and PCC target different problems. CST does not have the "proof carrying" aspect of PCC, but interestingly has a "code carrying" aspect that enables the verification.

Our work has connections with logic-based access control. Research on access control logic focus on expressiveness, decidability and theorem-proving efficiency of different logic frameworks. Lampson et al. defined a decidable logic based on the "*speaks for*" relation [17]. Appel and Felten found that many access control scenarios need higher-order logic, which is more expressive, but usually undecidable. They proposed proof-carrying authentication (PCA) [3], motivated by the idea of PCC, to shift the proof obligation to requestors. Code-carrying authorization (CCA) [18] is a follow-up of PCA. CCA allows requestors to provide fragments of the reference monitor's code (in form of the spi calculus), rather than proofs as in PCA. Our work is different from prior work on access control logic in two ways: (1) The certifier in a CST system is not a reference monitor; rather the computation being certified by the certifier is akin to a reference monitor; (2) The notion of proof in a CST system is partitioned into reasoning about trust (in the synthesizer) and logical correctness (in the program verifier), enabling the use of off-the-shelf program verifiers. On the other hand, proof systems for access-control are monolithic and based on custom axioms and inference rules about trust and authority, which makes it difficult to use off-the-shelf verifiers.

Connections can also be drawn between CST and secure multiparty computation [31] and verifiable computation [14] in applied cryptography. However, the goal are very different from CST. Secure multiparty computation is to enable parties to jointly compute a function over secret data held by individual parties. Verifiable computation enables a weaker device to securely outsource computations to untrusted servers.

VIII. Final Remarks

We show that CST is a practical approach for real developers to utilize program verification technologies to guard against logic flaws.

CST represents a paradigm shift for developers. Programming is less about conforming to a protocol, but more about explicating the computations in order to establish an end-to-end global safety property. From the security standpoint, protocols become advisory rather than mandatory. What is truly mandatory is the ambient predicates independent of these protocols.

Looking forward, we believe that the CST approach will get benefits by involving a broader community of protocol committees and API-providing companies. Today, the security goals are vague and confusing: protocols use different terminologies for same concepts; the global security goals are often implicit or buried in the step-by-step instructions of individual protocols. A valuable effort we envision is that the community agree on common terminologies and draft a "meta-specification" to formally define ambient predicates for each *class* of protocols, which will enable developers to do verification with little arbitrariness. As shown in the paper, this appears to be achievable in reality.

Acknowledgment

We thank our shepherd Mike Hicks and anonymous reviewers for insightful feedbacks and suggestions for improving the paper. We also thank Martín Abadi, Tom Ball, John Douceur, Úlfar Erlingsson, Cormac Herley, Jaeyeon Jung, Jay Lorch, Madan Musuvathi, Ed Nightingale, Prateek Saxena, Nik Swamy, Helen Wang and Lidong Zhou for commenting on an earlier version of this paper, Akash Lal and Mike Barnett for help on Corral and BCT, Rich Draves, Yuri Gurevich, Jon Howell, Mike Jones, and Yi-Min Wang for discussing the ideas. Eric Chen was supported in part by the Microsoft Research internship program.

References

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

[2] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. Proceedings of the 23rd IEEE Computer Security Foundations Symposium, 2010

[3] Andrew Appel and Edward Felten. Proof-Carrying Authentication. ACM CCS 1999.

[4] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. NDSS 2013.

[5] Michael Barnett and Shaz Qadeer. "BCT: A translator from MSIL to Boogie." Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2012.

[6] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Nikhil Swamy. "Verified implementations of the information card federated identity-management protocol," ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2008.

[7] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, James J. Leifer. "Cryptographic Protocol Synthesis and Verification for Multiparty Sessions". IEEE Computer Security Foundations Symposium, 2009

[8] Bruno Blanchet. Security Protocol Verification: Symbolic and Computational Models. In First Conference on Principles of Security and Trust (POST'12) (Pierpaolo Degano, Joshua Guttman, eds.), Springer Verlag, volume 7215, 2012

[9] Boogie: An Intermediate Verification Language. http://research.microsoft.com/en-us/projects/boogie/

[10] Cloud Security Alliance. "The Notorious Nine – Cloud Computing Top Threats in 2013". https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf

[11] Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, Karthikeyan Bhargavan, James Leifer. "Secure Implementations for Typed Session Abstractions". IEEE Computer Security Foundations Symposium (CSF), 2007

[12] DotNetOpenAuth. http://dotnetopenauth.net

[13] Tom FitzMacken. Using OAuth Providers with MVC 4. http://www.asp.net/mvc/tutorials/security/using-oauth-providers-with-mvc

[14] Rosario Gennaro, Craig Gentry, and Bryan Parno. "Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers". CRYPTO 2010.

[15] Dick Hardt. "The OAuth 2.0 Authorization Framework (RFC 6749)". http://tools.ietf.org/html/rfc6749

[16] Akash Lal, Shaz Qadeer and Shuvendu Lahiri. "A Solver for Reachability Modulo Theories". Computer Aided Verification, 2012

[17] B. Lampson, M Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems, theory and practice. ACM Trans. Comp. Sys. 10, 4, Nov. 1992.

[18] Sergio Maffeis, Martin Abadi, Cedric Fournet, and Andrew D. Gordon. Code-Carrying Authorization. ESORICS 2008.

[19] Microsoft. ASP.NET Sample for Live Connect OAuth SSO. https://github.com/liveservices/LiveSDK/tree/master/Samples/Asp.net

[20] Microsoft Corporation. MSDN article - Contract Class. http://msdn.microsoft.com/en-us/library/dd264808(v=vs.110).aspx

[21] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. USENIX Symposium on Operating Systems Design and Implementation, 1996

[22] Nieuwenhuis, R.; Oliveras, A.; Tinelli, C. (2006), "Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)", Journal of the ACM 53 (6), pp. 937–977.

[23] NopCommerce. http://www.nopcommerce.com/

[24] OpenID Foundation. "OpenID Authentication 2.0 - Final". http://openid.net/specs/openid-authentication-2_0.html

[25] Giancarlo Pellegrino and Davide Balzarotti. "Toward Black-Box Detection of Logic Flaws in Web Applications." Network and Distributed System Security (NDSS) Symposium, 2014

[26] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, E. Jay. "OpenID Connect Standard 1.0 - draft 21". http://openid.net/specs/openid-connect-standard-1_0.html

[27] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. *IEEE Symposium on Security and Privacy*, 2011.

[28] Rui Wang, Shuo Chen, XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. *IEEE Symposium on Security and Privacy*, 2012.

[29] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. *USENIX Security*, 2013.

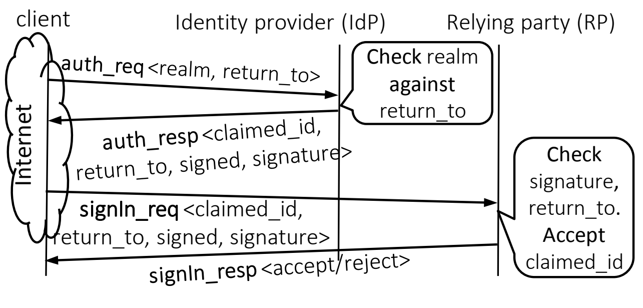
[30] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen, InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations, in Network & Distributed System Security Symposium (NDSS), February 2013

[31] Andrew Chi-Chih Yao: Protocols for Secure Computations (Extended Abstract). FOCS 1982

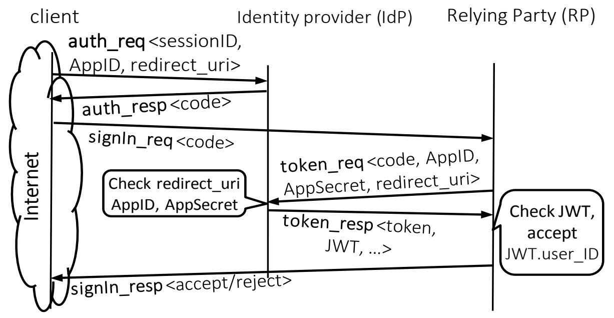
[32] A collection of online services enhanced by CST. https://sites.google.com/site/symbolictransaction/

Appendix A: OpenID-2.0 SSO and Live Connect SSO

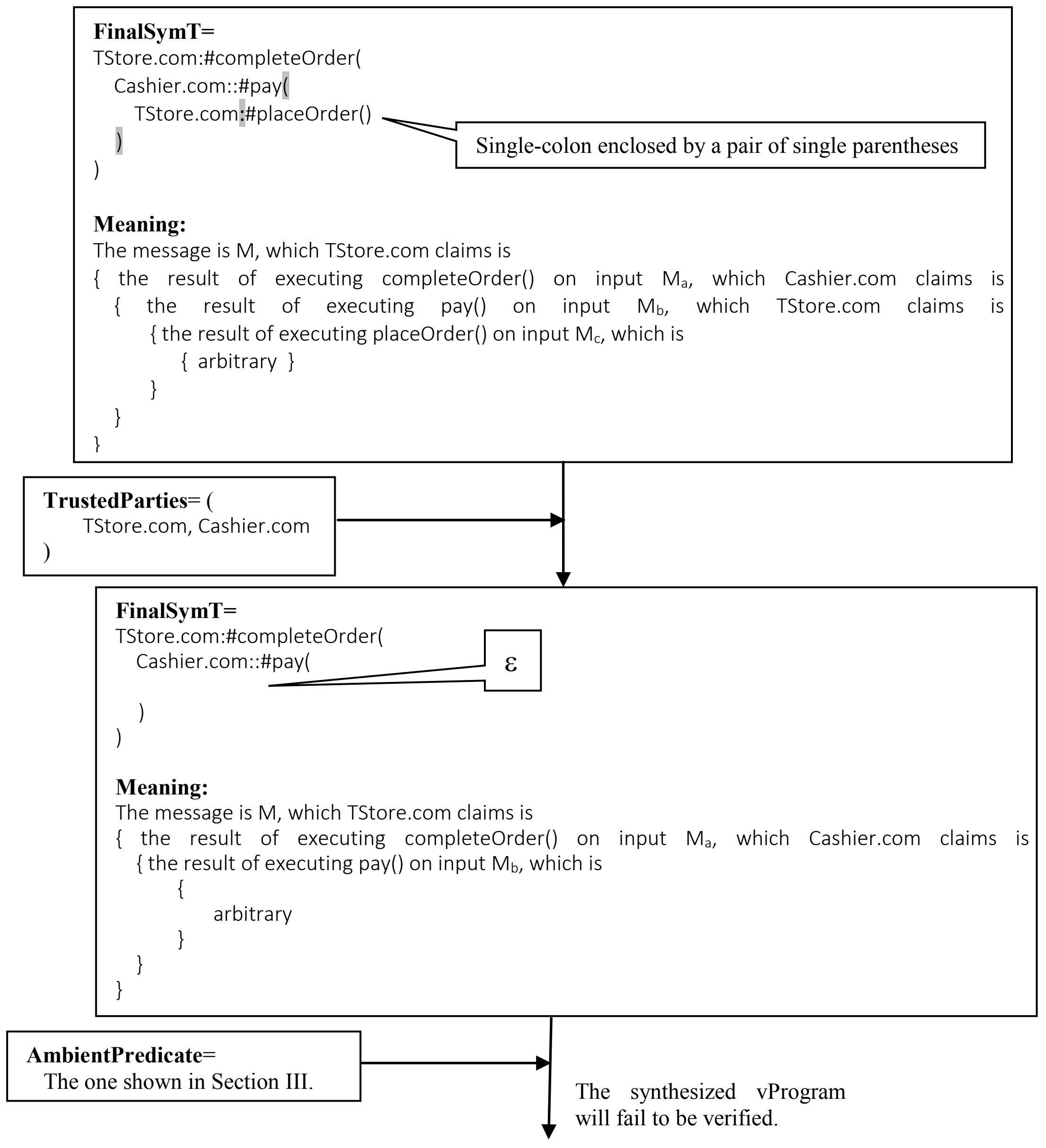
OpenID 2.0


client
Identity provider (IdP)
Relying party (RP)
Internet
auth_req <realm, return_to>
Check realm against return_to
auth_resp <claimed_id, return_to, signed, signature>
signIn_req <claimed_id, return_to, signed, signature>
Check signature, return_to. Accept claimed_id
signIn_resp <accept/reject>

Live Connect


client
Identity provider (IdP)
Relying Party (RP)
Internet
auth_req <sessionID, AppID, redirect_uri>
auth_resp <code>
signIn_req <code>
Check redirect_uri AppID, AppSecret
token_req <code, AppID, AppSecret, redirect_uri>
token_resp <token, JWT, …>
Check JWT, accept JWT.user_ID
signIn_resp <accept/reject>

Appendix B: An example of the synthesizer discarding an untrusted call
(Note that the single colon with the placeOrder call)



FinalSymT=

```
TStore.com:#completeOrder(
  Cashier.com::#pay(
    TStore.com:#placeOrder()
  )
)
```

Single-colon enclosed by a pair of single parentheses

Meaning:
The message is M, which TStore.com claims is
{ the result of executing completeOrder() on input M_a, which Cashier.com claims is
  { the result of executing pay() on input M_b, which TStore.com claims is
    { the result of executing placeOrder() on input M_c, which is
      { arbitrary }
    }
  }
}

TrustedParties= (
TStore.com, Cashier.com
)

FinalSymT=

```
TStore.com:#completeOrder(
  Cashier.com::#pay(

  )
)
```

ε

Meaning:
The message is M, which TStore.com claims is
{ the result of executing completeOrder() on input M_a, which Cashier.com claims is
  { the result of executing pay() on input M_b, which is
    {
      arbitrary
    }
  }
}

AmbientPredicate=
The one shown in Section III.

The synthesized vProgram will fail to be verified.