Self-Verifying Execution (Position Paper)

Matt McCutchen
MIT
matt@mattmccutchen.net

Daniel Song
Rice University
dwsong@rice.edu

Shuo Chen and Shaz Qadeer
Microsoft Research
[shuochen,qadeer]@microsoft.com

***Abstract*—This paper proposes a notion called *self-verifying execution (SVX)*. SVX substantially lowers several hurdles that real-world programmers face when adopting traditional program verification approaches. The current focus of SVX is to verify safety properties for programs that implement cloud-API integrations. We envision that, if adopted by real-world programmers, the SVX approach will enable a positive paradigm shift in the community toward more rigorous reasoning about security goals of cloud-API protocols.**

Keywords—program verification; self-verifying execution; cloud-API integration; single-sign-on (SSO)

I. INTRODUCTION

Program verification, if widely adopted by real-world programmers, would be a strong approach to ensure system safety and security. Unfortunately, verification technologies are usually too demanding for most programmers. As a result, they are rarely adopted in the real world.

In this paper, we propose a notion called *self-verifying execution* (SVX). The core idea is that every actual execution in the system is responsible for collecting its own executed code, and symbolically proving that the code satisfies all desired safety properties. Essentially, SVX is proving program code on a per-execution basis at runtime, whereas traditional approaches try to verify a priori the safety properties for all possible executions. In this sense, SVX falls into the general notion of runtime verification [8]. However, SVX's amortized runtime cost is near zero, because all theorems proven by SVX are symbolic and can be effectively cached.

SVX addresses several classic hurdles that programmers face when applying program verification in the real world. These hurdles include: (1) the need for precise modeling of the client's behaviors and the runtime platform and (2) the need to analyze executions of unbounded length, which is hard to automate. SVX substantially lowers these hurdles, because (1) programmers don't need to model most aspects of the client's behaviors or the runtime platform, since every execution to be verified is driven by a real user on a real platform, and (2) the proof obligation is significantly lowered – the theorem to prove is only about a set of executions similar to the current execution, not about all possible executions. Furthermore, the designer of a protocol can write an abstract base class that sets up all the necessary interaction with the SVX framework (including the desired safety properties), and all concrete subclasses will be automatically verified against the same properties without the *end programmers* who write them having to know anything about SVX.

For these reasons, we believe that the SVX-style verification is practical for real-world programmers in certain application domains.

Current focus of SVX. An application domain in which we are applying SVX is the security of cloud-API integration. Many major companies provide services as cloud APIs. These services include single-sign-on (SSO), online payment, social sharing, cloud storage, etc. They are integrated into millions of websites and mobile apps [7]. However, the current practice is fairly ad-hoc – basically, programmers just read protocol specifications and developer's guides to implement their code, but there is no assurance that the implementations meet the security goals of the protocols (admittedly, the security goals themselves are unclear in the protocol specifications). Studies have shown many logic bugs in real websites and apps that can cause serious security breaches. For example, an attacker can sign into other people's accounts or make purchases without paying [4][9][10] [12][13][14]. This type of issue is ranked by the Cloud Security Alliance as the No.4 cloud computing top threat [5].

To fundamentally solve the problem, it is important to bring rigor into the practice of cloud-API integration. We are making an effort in this direction, and SVX is a key enabler. Section III will explain our open-source framework that incorporates SVX into an object-oriented design for SSO solutions. We envision that protocol designers, service-providing companies and end programmers write code at different abstraction levels in this single codebase. SVX is able to ensure that every concrete app implementation satisfies the properties that protocol designers specify. The framework can accommodate most major SSO solutions. We have demonstrated a variety of implementations that integrate Microsoft, Facebook, Google, Yahoo SSO services, which are based on OpenID 2.0, OAuth 2.0 and OpenID Connect 1.0 protocols. All implementations are verified against a protocol-independent safety property with no effort by the end programmer.

We hope that the real-world adoption of SVX will enable a paradigm shift so that our community puts more emphasis on end-to-end properties, rather than step-by-step instructions, when specifying cloud-API protocols.

This position paper describes the research direction consisting of our published work [4] with several important recent improvements. The differences from [4] are summarized in Section IV.

II. Traditional Verification vs. SVX

A. *Program verification is a demanding task*

Program verification in the real world is more than simply feeding a program "P" and a property "φ" into an automatic verifier. It is much more demanding. First, the program runs on an underlying system platform, which is often called *the environment* "E" in the software-testing terminology. An effective model for E needs to be constructed for the verification task. Second, the program is driven by an entity whose behavior is arbitrary (e.g., a user, an app, a client, etc.). The entity needs to be modeled as *a test harness* H that can trigger all possible execution paths. For a security problem, modeling E and H is especially difficult because (1) H is the attacker, whose behaviors are hard to anticipate exhaustively, (2) it is also hard to objectively determine which aspects of the environment E "matter" and which are "details" to abstract away when considering a given security problem.

Even if E and H are precisely modeled, verifying property φ is still challenging. Usually P exposes several public methods that H (i.e., the attacker) can call arbitrarily. A traditional verification approach needs to prove that φ is satisfied even if H makes an infinite number of such calls (i.e., H must be modeled as an outermost infinite loop that invokes P's methods). Real-world code is typically complex enough that verification requires invariants and/or lemmas to be specified by the programmer, which is not realistic for a typical end programmer who wants to integrate cloud APIs into a website.

B. *Basic idea of SVX*

SVX is an approach to lower these hurdles so that normal programmers can build a system that is verified against a safety property. We use a simple example, called the "ABC system", to explain the idea. The system consists of three websites *Alice.com*, *Bob.com* and *Charlie.com*, as shown in Figure 1. Each website holds an integer constant and has a public method. A client can chain together an arbitrary sequence of calls to these methods by starting with an arbitrary instance of class Message and feeding the output of a method call into the next method call in the chain. max(m1,m2) compares the two input objects based on the value field and returns the bigger one. The system needs to ensure that when conclude(m2) is called, m2.largestParty should indicate the party holding the largest value of the three, i.e., property φ below must be satisfied:

```
((m2.largestParty == "Alice") ==>
    (Alice.value >= Bob.value ∧ Alice.value >= Charlie.value))
  ∧ ((m2.largestParty == "Bob") ==>
    (Bob.value >= Alice.value ∧ Bob.value >= Charlie.value))
  ∧ ((m2.largestParty == "Charlie") ==>
    (Charlie.value >= Alice.value ∧ Charlie.value >= Bob.value))
```

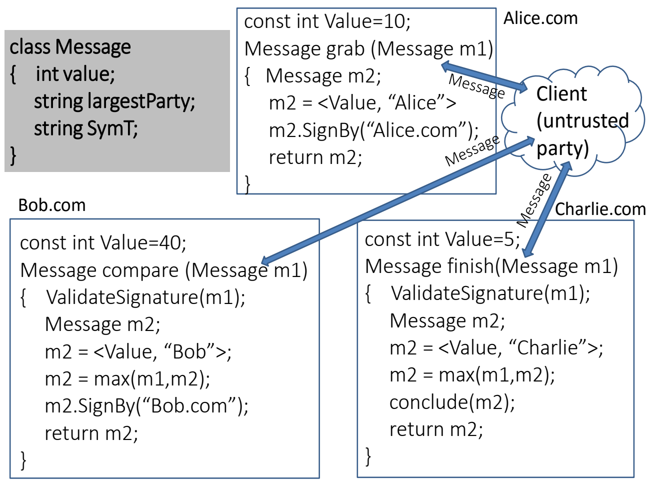
There is an extra field SymT in class Message whose purpose will be explained shortly.

Threat model. The threat model we consider is the *web attacker model* [2]. We require that all messages are sent over HTTPS, so threats of network attackers (e.g., routers or sniffers) are not of our concern. Also, we consider lower-level language bugs (e.g., buffer overrun or cross-site scripting) orthogonal to the problem we target. There is a rich body of literature addressing low-level issues.

Basic idea of SVX. Unlike traditional approaches, the goal of SVX is to verify at runtime if *the current execution satisfies φ*. The execution in Figure 2 is correct. We now show how SVX verifies this execution by leveraging the field SymT in class Message. SymT, the "*symbolic transaction*" of the current execution, is a string that can be thought of as an onion: for the execution in Figure 2, SymT is initially an empty string ε, representing a non-deterministic input message; as the execution goes on, the SymT in each message wraps the newly executed method over the previous SymT. Thus, SymT essentially denotes how the result <40,"Bob"> is obtained through the execution (substrings #grab, #compare and #finish represent *method hashes*, which will be explained in Section II.C). In the end, the message passed to conclude() is the following.

```
< 40,"Bob",
  Charlie.com:#finish(Bob.com::#compare(Alice.com::#grab())) >
```

When conclude is called, SVX verifies that the code sequence represented by m2.SymT logically implies φ. The



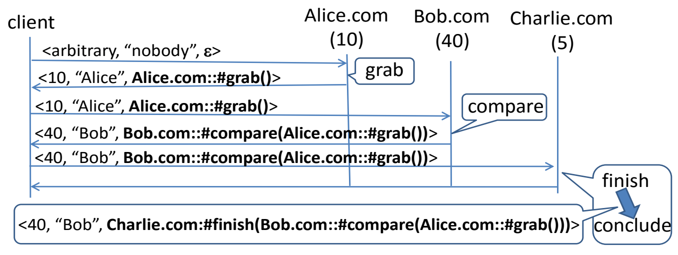
```
class Message
{   int value;
    string largestParty;
    string SymT;
}
```

```
const int Value=10;
Message grab (Message m1)
{   Message m2;
    m2 = <Value, "Alice">
    m2.SignBy("Alice.com");
    return m2;
}
```

```
const int Value=40;
Message compare (Message m1)
{   ValidateSignature(m1);
    Message m2;
    m2 = <Value, "Bob">;
    m2 = max(m1,m2);
    m2.SignBy("Bob.com");
    return m2;
}
```

```
const int Value=5;
Message finish(Message m1)
{   ValidateSignature(m1);
    Message m2;
    m2 = <Value, "Charlie">;
    m2 = max(m1,m2);
    conclude(m2);
    return m2;
}
```

Figure 1: The ABC system.



client
Alice.com (10)
Bob.com (40)
Charlie.com (5)
<arbitrary, "nobody", ε>
<10, "Alice", Alice.com::#grab()>
grab
<10, "Alice", Alice.com::#grab()>
<40, "Bob", Bob.com::#compare(Alice.com::#grab())>
compare
<40, "Bob", Bob.com::#compare(Alice.com::#grab())>
finish
<40, "Bob", Charlie.com::#finish(Bob.com::#compare(Alice.com::#grab()))>
conclude

Figure 2: A concrete execution of SVX.

verification is called *certification* in SVX. For the execution in Figure 2, the certification will succeed, so the execution leading to the result <40,"Bob"> is secure and accepted. On the other hand, if the client calls grab on Alice.com and then immediately calls finish on Charlie.com, Charlie's conclusion is not the overall maximum, but only the maximum of Alice's and Charlie's values. For this execution, the last message will be <10,"Alice",Charlie.com:#finish(Alice.com::#grab())>. The computation represented by this SymT is insufficient to ensure φ, so the certification fails and the execution is rejected.

While this paper discusses only linear method sequences, we have extended SVX to support methods with multiple input messages and to automatically detect the most common cases in which the same message is used more than once in a SymT, so in general a SymT describes a directed acyclic graph of data flow with some restrictions.

The runtime overhead of SVX may seem prohibitively high, because the code sequence of every execution is verified against property φ. Fortunately, because the SVX verification is only about code, not concrete data, the proven theorems can be effectively cached. Hence, every execution only needs to pay a cache-lookup cost, unless its code sequence is not seen before.

Trust relations in SymT. Each property verified with SVX must specify a set of *trusted parties*, which are assumed never to add false information to a SymT. These parties should be the ones on whom the property inherently depends (i.e., it cannot be proved without some knowledge of their behavior) and are the same ones that would be modeled as known programs rather than nondeterministic agents in traditional offline verification. Because an execution goes through different parties, trusted or untrusted, the SymT representation denotes different trust relations: (1) unsigned and signed messages are denoted using single-colon and double-colon, respectively; (2) browser redirections vs. server-to-server calls are denoted using single-parentheses and double-parentheses, respectively. Many details about trusting the SymT "onion" are described in Section IV.A of our paper [4]; the following paragraph gives a sketch.

When the SymT for the execution in Figure 2 is presented for certification, the certifier examines it like an onion: the SymT is scanned inwards, starting from Charlie.com:#finish. As soon as the certifier encounters an untrusted layer, which is either an unsigned browser redirection or an untrusted party (e.g., an unknown *David.com*), the layer and everything inside are discarded from the SymT. For example, if *Bob.com* forgot to sign its message, then the second layer of SymT would be "(Bob.com:#compare…)", with single-parentheses enclosing a single-colon (i.e., an unsigned browser redirection). The certifier would discard it with everything inside, so the SymT would be treated as "Charlie.com:#finish()", which denotes that *Charlie.com* runs finish() on an arbitrary input. It would not pass the certification, and the execution would be rejected.

C. *Public interface and internal mechanism of SVX*

We have built SVX as a library that exposes only two public functions: RecordMe and Certify. For the "ABC system", programmers add m2.SymT=RecordMe(m1.SymT) inside methods grab, compare and finish and call Certify(m2.SymT, φ) inside method conclude. RecordMe uses the language's reflection capability to compute the *method hash* of the caller method, and concatenate it with m1.SymT with colons and parentheses to form m2.SymT. The method hash is computed using SHA-1 over the following pieces of information of the caller method: the method name, the class name of the concrete object of the method, the code of the containing DLL, and the class names of the input/output messages. Recording the class names is important – it allows the SVX mechanism to be built into an abstract base class which can record executions on every concrete implementation. Note that the "ABC system" does not have the aspect of abstraction/concretization. The aspect will become clear when we discuss SSO protocols and implementations in Section III.

Function Certify is shown in Figure 3. First, it performs the scanning operation described earlier in order to discard the untrusted "onion core" in the SymT, if any. The resulting SymT represents the computational sequence that we trust. This SymT and property φ constitute the theorem to be proven. There is a theorem cache to store the theorems that have been examined before. If the current theorem exists in the cache, the cached result (i.e., pass or fail) is returned. Otherwise, the theorem is passed to a remote server, called the *certification server*. The server does two things: first, it synthesizes a program to represent the computations recorded in the SymT. The synthesized program is called the *vProgram*. This step requires a "de-hash" table, which can retrieve method information and DLL code corresponding to a method hash. Second, it verifies the vProgram using an off-the-shelf program verifier, described in our paper [4]. We currently host the certification server in the Microsoft Azure cloud.

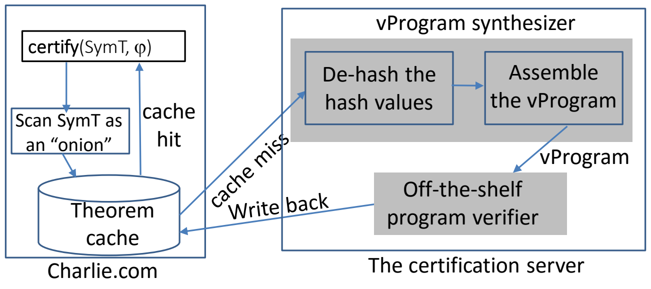


Figure 3: Function Certify and the certification server.

D. *Limitations of SVX*

Monotonicity of properties. In principle, SVX can only verify properties of the form "every time this point in the code is reached, φ holds", where φ is *monotonic*, i.e., one can deduce from a recorded method sequence that φ holds regardless of what additional events may have occurred in the system outside the current transaction. In our implementation, φ is a C# expression that is evaluated on the final state of the vProgram, so in order for it to be interpretable in terms of the real system, some relationship must be maintained between the states of the vProgram and the real system. Typically, φ asserts the presence of entries in append-only data structures stating that a party determined a certain condition to be met (e.g., the identity provider received the correct password from a user); if the entries are present in the vProgram, they will also be present in the real system. A future version of SVX will enforce that φ takes this form. Ensuring that the implementation generates the entries only under the right circumstances is outside the scope of SVX. The closest one can come to asserting a nonmonotonic property such as "the user's access has never been revoked" is to have one party generate an entry stating that it had no revocation on record as of a specific time and then have φ assert the presence of a sufficiently recent entry, effectively moving the nonmonotonicity outside the scope of SVX (but still verifying any subsequent protocol logic).

Modeling. SVX cannot remove the need to model any aspects of the environment and/or the attacker that affect the verification of a *single, known* SymT. For example, the security of SSO protocols is generally based on secret values exchanged between websites, so a comprehensive verification of such a protocol (going beyond the necessary condition discussed in III.B below) must somehow model the attacker's knowledge of these secrets, whether or not SVX is used. However, typically much of the "boring" complexity of the system (such as the mapping of HTTP requests to handlers) becomes irrelevant when a single SymT is given.

III. Current Application area of SVX

In the introduction, we explained the ad-hoc nature of today's cloud-API integration, which results in many logic bugs in real websites. We believe that SVX is a promising technology to fundamentally address these logic bugs. Specifically, we are working on an open-source project named SVAuth, which is an object-oriented framework for protocol designers, service-providing companies and end programmers to build single-sign-on (SSO) solutions in a common codebase. The goal of SVAuth is to ensure that every concrete SSO implementation that an end programmer constructs satisfies the safety properties that the protocol designers intend.

A. *Overview of SVAuth*

SVAuth is implemented using C#. It runs on *.NET Core* [1], which is a light-weight .NET runtime for Windows, Linux, OS X and Docker. An important aspect about SVX, which was not shown in the "ABC system", is how it works with abstraction and concretization.

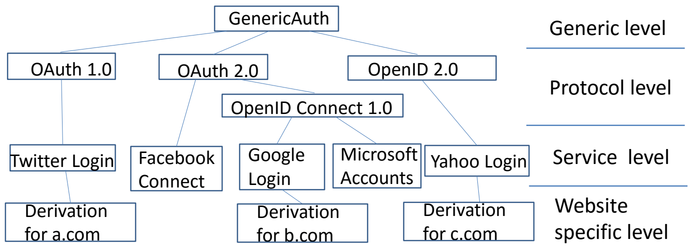


Figure 4: The class hierarchy of SVAuth

Figure 4 shows the class hierarchy of the SVAuth framework. It consists of four levels. The top level is called "GenericAuth", which defines the classes for the most basic concepts in SSO, including identity provider (IdP), relying party (RP) and common messages in all SSO solutions. Moreover, it defines the necessary safety properties that all SSO executions should satisfy.

GenericAuth does not specify how the messages are exchanged and handled, which is defined by the protocol-level classes corresponding to each individual protocol, such as OAuth 2.0 and OpenID 2.0. The service level consists of classes that implement solutions for Facebook, Google and other service companies. These classes are derived from protocol-level classes. Note that every company's cloud APIs allow several reasonable solutions to be implemented, so a box in Figure 4, e.g., Google Login, only represents *a solution*, not *the solution* for Google login. At the fourth level, individual websites may derive from the service level to adapt the solutions to their specific needs.

B. *Enabling SVX in the class hierarchy*

The goal of SVAuth is to ensure that all concrete websites satisfy the safety properties specified at the GenericAuth level. Traditional approaches accomplish this goal via refinement [11], i.e., proving that the GenericAuth-level properties are satisfied at every level in the hierarchy, an approach that demands a lot from end programmers. Instead, since the proof obligations generated by SVX are simple enough to be solved automatically, we just solve them separately for each concrete implementation. We build the self-verifying capability into the top two levels of the class hierarchy; SVX ensures that all concrete websites inherit the capability automatically. Specifically, we simply call Certify at the GenericAuth level, and call RecordMe in every message handler at the protocol level. Programmers at the service level and the website specific level do not need to be aware of the SVX mechanism. The only expertise they need is object-oriented programming. In the rest of this subsection, we present more details about our approach.

Safety property φ. Our previous paper [4] defines a safety property that is necessary for SSO (in Section V.B on page 9 of the paper). It is defined jointly over the verifying RP and the IdP that this RP wishes to use, which are the trusted parties (compare to the property in the ABC example, which is defined over Alice.com, Bob.com and Charlie.com). Intuitively, the property asserts that: when the RP has made the conclusion that the client is "Alice", then this SSO execution should have witnessed an *ID claim* on the IdP in which the user ID is "Alice", the realm equals this RP's identifier, and the *ID claim* is redirected to the web address of the RP. To give a flavor of the property's definition, we show the C# assertion below. Due to the space constraint, we omit a detailed explanation of how GenericAuth defines the SSO core concepts.

```
ID_claim = IdP.IdentityRecords.getEntry(
                SignInIdP_Req.IdPSessionID, RP.Realm);
Contract.Assert(ID_claim.Redir_dest == RP.Domain
                && ID_claim.UserID == conclusion.UserID);
```

It is important to note that this property is just one necessary condition of SSO. There are other conditions that should be defined for SSO security. Defining properties to capture all real-world security concerns is an effort that needs to involve a broader discussion in the SSO community. The current focus of SVAuth is to build the underlying verification technology so that if these properties can be defined, they can be verified.

An example implementation. Figure 5 shows one of our demo websites (called foo.com), which uses Microsoft Accounts login. The GenericAuth level defines the parties (e.g., GenericAuth.IdP) and messages (e.g., SignInIdP_req). It also defines the safety property using these parties and messages. The OAuth 2.0 protocol inherits from GenericAuth, and the OpenID Connect 1.0 (a.k.a. OIDC 1.0) inherits from OAuth 2.0. The protocol-level classes define message handlers, such as ConstructTokenRequest in OIDC 1.0. Each message handler calls RecordMe to compute the SymT string. These handlers call some virtual methods which need to be concretized by the service level for Microsoft, Facebook, Google, etc., and by concrete websites like foo.com. At the bottom two levels, programmers do not call either RecordMe or Certify, and only need to concretize virtual methods defined by the protocol level.

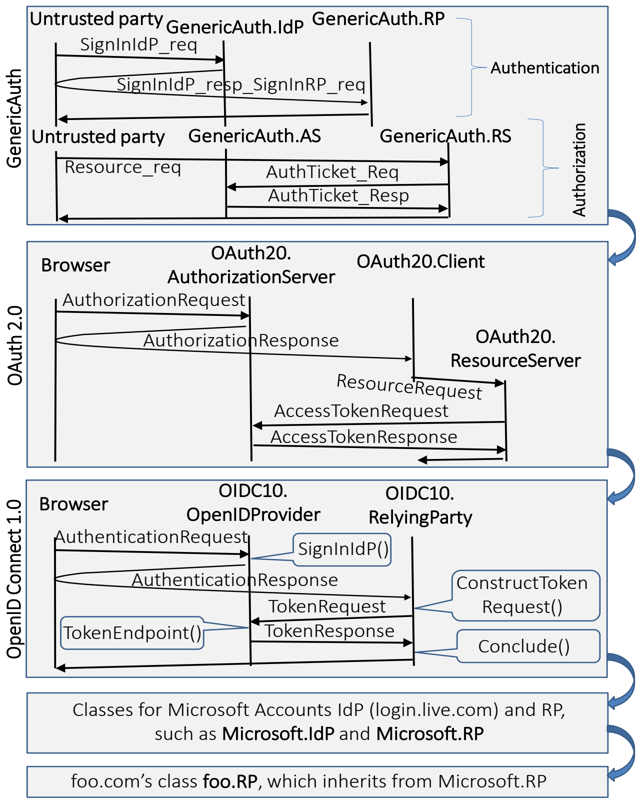


Figure 5: Website foo.com using Microsoft Accounts login.

During every execution, RecordMe records the concrete class of the caller object. In other words, when ConstructTokenRequest calls RecordMe, the resulting method hash captures the fact that the class of the concrete object is foo.RP, not just OIDC10.RelyingParty. When an execution goes through the entire flow, the final SymT will be: foo.com:#Conclude$_{\text{foo.RP}}$((Live.com:#TokenEndpoint$_{\text{Microsoft.IdP}}$((foo.com:#ConstructTokenRequest$_{\text{foo.RP}}$(Live.com:#SignInIdP$_{\text{Microsoft.IdP}}$()))))), in which the subscripts denote the classes of the concrete objects. If the same OIDC10 protocol flow is concretized by a website bar.com and the Google login service, then the SymT will lead to a different theorem to verify. This essentially means that, without using the approach of refinement, protocol designers can spend a one-time effort building SVX into the protocol-level classes, so that the effort will be massively scaled up to cover all derived implementations.

C. *Real-world deployment*

In this subsection, we explain how SVAuth will be deployed by programmers in the real world.

Platform independence. Real-world websites are built on different platforms, such as PHP, JSP, ASP.NET, Python, Node.JS, and many others. The current situation is that every identity service company, such as Facebook, Google, and Microsoft, publishes libraries for some of the platforms; for other platforms, programmers have to search for suitable libraries from the web or call the raw APIs according to protocol specifications. In fact, even with a library, the integration may not be easy, because different libraries expect different ways of integration, so programmers need to understand the documentation and sample code for each library.

SVAuth provides a platform-independent solution for every web programmer. As explained earlier, it is written in C# and runs on .NET Core as a standalone web service,

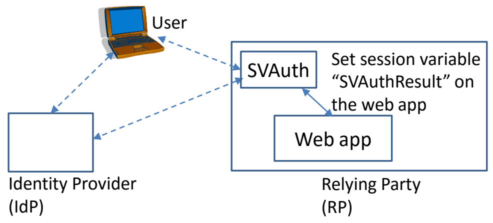


Figure 6: SVAuth is small standalone executable on the RP.

listening on its own port, as opposed to a library to be linked with a web app. Figure 6 shows SVAuth in an SSO system. The dashed-line triangle in the figure performs the SSO execution without involving the web app. The web app can be built in any web platform, not necessarily .NET Core.

The integration of an SSO solution into the web app is as easy as pasting a line of code. Every SSO solution in SVAuth is represented as a button, such as “Log In”. The source code (which is a single line of code) of the button can be obtained by the programmer by right-clicking the button. To integrate the solution, the programmer only needs to paste this line of code into her web app. Consequently, when a user clicks on the button, the entire SSO execution will be accomplished by SVAuth, and the result will be set to the web app’s session variable “SVAuthResult”. Therefore, the web app does not need to be aware of the SSO protocol. The entire SSO functionality is just like a local function – the button calls the function, and the result will be in the session variable.

Involving identity service companies. The SVX approach expects every web service in an execution to participate in computing the SymT string. For SSO, this means that identity service companies like Facebook, Google and Microsoft need to provide method hashes in their responses.

We envision two stages to make this happen. In the current (first) stage, the companies are not aware of the SymT field. Hence, the current version of SVAuth has to provide source files that model the IdP’s methods. Every call to an IdP is sent to the actual IdP, but the method hash of the response is computed using the corresponding method in the IdP’s model. Of course, the caveat is that the model is only our best-effort approximation for the actual IdP’s logic.

In the second stage, we will reach out to the identity service companies so that they can review and correct our current models. Eventually, if the companies are persuaded to attach the method hashes by themselves, the SVAuth codebase will not need the IdP models. Services that are not implemented in .NET can still maintain their own .NET model and attach the corresponding hash, so other SVX-enabled services can interoperate with them transparently.

D. *Runtime performance*

TABLE I shows the performance numbers that we obtained using a Windows 8.1 machine with a 2.5GHz CPU and 16GB RAM. The measured implementations include the ABC system, and our RP implementations interacting with Microsoft (LiveID) login, Yahoo login and Facebook login.

As explained earlier, the computed method hashes and certified SymTs are cached, so the actual runtime overhead is near zero. Specifically, a method needs to compute its hash only in its first run after a website starts; the vProgram synthesis/verification happens only when an execution goes through a previously unseen sequence of methods or any DLLs containing recorded methods were recompiled. In either scenario, the overhead is a one-time cost. The per-execution cost for recording SymT is simply a set of string concatenation operations, and that for certifying a SymT is a string scanning (for the onion) and a local cache lookup. As a comparison, the table provides the breakdown of the one-time overhead, which is indeed expensive. It includes the compilation overhead, and the overheads of RecordMe and Certify if caching was disabled.

IV. RELATED WORK

Unlike the SVX approach, offline (i.e., completely static) verification approaches were used by researchers to analyze SSO protocols and implementations. For example, Bansal et al. [3] used ProVerif to analyze OAuth 2.0. ProVerif uses sound approximations for executions of unbounded length, which may in general introduce false positives. This work was based on handwritten models in the applied pi-calculus, not implementation code. It took a long time to run the verification (e.g., almost 3 hours for verifying the OAuth 2.0 authorization code flow). Fett et al. constructed a formal model for the BrowserID SSO protocol, and used it to guide their security investigation about the protocol.

The most closely related work is our earlier paper [4]. However, the current paper has three main aspects representing new development: (1) the earlier paper did not have the aspect of object-oriented class hierarchy, in which SVX is built into abstract base classes and automatically inherited by all concrete implementations; (2) we did not use

TABLE I: RUNTIME OVERHEAD – PER-EXECUTION AND ONE-TIME COSTS.

	Protocol party	**Per-execution cost**	**One-time cost**		
		Runtime overhead	vProgram compilation	Average overhead of RecordMe without caching	Average overhead of Certify without caching
	Alice	≅ 0ms	670ms	247ms	
The ABC system	Bob	≅ 0ms	635ms	222ms	6666ms
	Charlie	≅ 0ms	646ms	231ms	
Microsoft (LiveID) login	IdP	≅ 0ms	852ms	151ms	
	RP	≅ 0ms	1178ms	228ms	13674ms
Yahoo login	IdP	≅ 0ms	678ms	153ms	
	RP	≅ 0ms	1073ms	165ms	10959ms
Facebook login	IdP	≅ 0ms	777ms	163ms	
	RP	≅ 0ms	1001ms	205ms	11968ms

the language's reflection capability, but required a programmer to manually copy source code of invoked methods as string constants for vProgram synthesis; (3) it was not built on .NET Core runtime, and was a library only usable for ASP.NET websites.

V. Summary and Our Vision

This paper describes the basic idea of SVX, which lowers the hurdles that real-world programmers face when applying program verification. Specifically, we explain how to apply the SVX strategy in the area of online-API integration. We propose the SVAuth framework, which is able to ensure that every SSO execution satisfies safety properties that protocol designers intend. It seems promising that all major SSO solutions can be built within the framework. Because SVAuth is platform independent, it will benefit all web platforms.

The vision of SVAuth is that protocol designers, service companies and individual website programmers can work together on a single codebase, so that the SSO security of every concrete website can be checked end-to-end. The use of SVAuth will enable a paradigm shift in the community that leads to two positive impacts: (1) People will no longer think of a protocol specification as an English document with pseudo-code examples. Instead, a specification is a set of abstract classes that comprises real code running on every website. (2) The SSO community today spends most effort specifying step-by-step instructions for programmers, but no real effort specifying end-to-end safety properties that the protocols must achieve. Programmers often pay much attention trying to understand protocol-specific details, yet fail to build secure implementations. The SVAuth effort can motivate the community to define end-to-end properties, in addition to step-by-step procedures for achieving those properties.

References

[1] .NET Core. https://www.microsoft.com/net/core

[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] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. Journal of Computer Security, Vol. 22, No. 4, July 2014, pp. 601-657.

[4] Eric Chen, Shuo Chen, Shaz Qadeer, and Rui Wang. Securing Multiparty Online Services via Certification of Symbolic Transactions, IEEE Symposium on Security and Privacy 2015.

[5] 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

[6] Daniel Fett, Ralf Küsters, and Guido Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. IEEE Symposium on Security and Privacy, 2014.

[7] SimilarTech. Number of websites using Facebook Connect. https://www.similartech.com/technologies/facebook-connect

[8] The Runtime Verificaiton Workshop. http://runtime-verification.org/

[9] Fangqi Sun, Liang Xu, Zhendong Su. Detecting Logic Vulnerabilities in E-Commerce Applications. Proceedings of The Network and Distributed System Security Symposium 2014

[10] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. ACM conference on Computer and Communications Security 2012.

[11] Niklaus Wirth. Program Development by Stepwise Refinement. Communications of the ACM, Vol. 14, No. 4, April 1971, pp. 221-227.

[12] 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

[13] 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.

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