Self-Verifying Execution

Matt McCutchen  Daniel Song  Shuo Chen, Shaz Qadeer
MIT  Rice University  Microsoft Research
Motivation

• Logic bugs are pervasive in online-service protocol implementations
  • Online-service protocols: single-sign-on, payment, authorization
  • Security consequence
    • signing into other people’s accounts without password
    • making purchase without paying
    • Unauthorized access
  • This bug category is ranked as the No.4 cloud security top threat.

• Program verification
  • If adopted by real-world programmers, it would fundamentally avoid logic bugs.
  • Unfortunately, too demanding for most programmers.

• The goal of self-verifying execution (SVX)
  • To substantially lower hurdles for real-world programmers to do verification.
Hurdles of traditional verification approaches

- Need to model the system platform
- Need to model the client behavior
- Need to prove an inductive theorem for all possible executions (an infinite set)

The proof obligation:
In the end of EVERY possible execution, \( \phi \) is satisfied.
Basic idea of self-verifying execution

- Every actual execution is responsible for collecting its own executed code, and symbolically proving that it satisfies $\varphi$.
Example: comparing integer constants among three websites

Safety property $\varphi$:
Whenever $\text{conclude}(m_2)$ is reached, $m_2$ must represent the website holding the biggest int.

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

#### Bob.com
```c
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;
}
```

#### Charlie.com
```c
const int Value=5;
Message finish (Message m1)
{
    ValidateSignature(m1);
    Message m2;
    m2 = <Value, "Charlie”>;
    m2 = max(m1,m2);
    m2 = max(m1,m2);
    m2.SignBy("Charlie.com”);
    return m2;
}
```
The expected protocol flow

client

<arbitrary, “nobody”>

<10, “Alice”>

<10, “Alice”>

<40, “Bob”>

<40, “Bob”>

Alice.com

grab

Bob.com (40)

compare

Charlie.com (5)

finish

<40, “Bob”>

conclude

(10)

(40)
The system is vulnerable!

Client

<arbitrary, "nobody">

<10, "Alice">

Alice.com (10) grab

finish

<10, "Alice">

Bob.com (40)

Charlie.com (5)

<10, "Alice">

conclude

x
How SVX works

• Attach a field, namely *SymT (Symbolic Transaction)* onto every message.

• `#grab`, `#compare` and `#finish` are the hash values of the executed code of these methods.
Verifying an execution

• Method conclude() calls a program verifier to prove:
  The final $\text{SymT} \rightarrow \varphi$
  
  • Charlie.com:#finish(Bob.com::#compare(Alice.com::#grab())) \rightarrow \varphi$, the execution is accepted.  
  • Charlie.com:#finish(Alice.com::#grab()) \not\rightarrow \varphi$, the execution is rejected.

• Note that the program verification is symbolic (only about code). The concrete values are ignored.
  • A middle ground between offline symbolic verification and runtime concrete checking.
The library we provide

• Only two public methods

  • RecordMe(...): to construct the SymT

  • Certify(...): to verify the execution represented by the SymT.
RecordMe() for SymT construction

• Just add a RecordMe() call in each message handler method.

• RecordMe() uses reflection to hash the current method code, and concatenates the new SymT.

```java
int Value=10;
Message grab (Message m1) {
    Message m2;
    RecordMe(m1,m2);
    m2 = <Value, “Alice”>
    m2.SignBy(“Alice.com”);
    return m2;
}

int Value=40;
Message compare (Message m1) {
    Message m2;
    RecordMe(m1,m2);
    m2 = <Value, “Bob”>
    m2 = max(m1,m2);
    m2.SignBy(“Bob.com”);
    return m2;
}
```

```java
int Value=5;
Message finish (Message m1) {
    ValidateSignature(m1);
    Message m2;
    RecordMe(m1,m2);
    m2 = <Value, “Charlie”>
    m2 = max(m1,m2);
    if ( ! Certify(m2.SymT, φ))
        throw new Exception();
    conclude(m2);
    return m2;
}
```
Certify(): to verify SymT against $\varphi$

**Diagram:**
- **Theorem cache** (Charlie.com)
- **C# verifier** (CSC+BCT+Corral)
- **The certification server on Azure**

**Steps:**
1. **certify(SymT, $\varphi$)**
2. Cache hit (≤ 0 ms)
3. De-hash the hash values
4. Assemble the program to be verified
5. C# verifier (CSC+BCT+Corral)
6. Cache miss (20 secs)
SVX lowers burdens for programmers

• Reduce the burden of modeling the attacker and the runtime platform

• The theorem (i.e., proof obligation) is much easier to prove automatically

• The self-verifying capability can be inherited
  • It is well known that a property proven for a base class may not carry through onto concrete classes. (Liskov Substitution Principle, LSP)
  • However, the self-verifying capability does carry through.
Focused project: SVAuth
Overview of the SVAuth project

• Problem
  • Logic bugs in website sign-on mechanisms
  • Just like incorrectly installed front door locks for websites

• Vision
  • Every website’s front door lock should be installed with proven correctness.

• What is SVAuth?
  • Based on SVX, we are building relying party solutions that cover all major identity services in the world.
    • We have built solutions for 5 out of 30 major identity services
Architecture of SVAuth

Identity provider (e.g., Facebook.com)

SVAuth (a local service running on .NET Core)

Web app (e.g., PHP)

Relying party Foo.com

The only interface: setSessionVars

The only interface: setSessionVars

.NET Core for C#

≈

Node.JS for Javascript

The only interface: setSessionVars

The only interface: setSessionVars

client

authentication protocol

The only interface: setSessionVars

The only interface: setSessionVars

The only interface: setSessionVars
SVAuth’s class hierarchy

Protocol-independent level
(defined a set of safety properties for all protocols)

Protocol level
(where RecordMe calls are added into message handlers)

SDK level and website level:
* Programmers just need to do normal OO programming.
* The self-verifying capability will be inherited automatically.
Summary

• SVX is a practical method to do verification for online protocol implementations

• SVAuth seems promising to achieve our vision
  • Every website’s front door lock should be installed with proven correctness.

• Realistic to use
  • Super easy to integrate SVAuth with real-world software
    • E.g., MediaWiki and HotCRP -- only a few lines of essential code changes.
    • A Microsoft Research website has been running MediaWiki with SVAuth for months

• You can help
  • If your website needs Facebook, Google, Yahoo, Microsoft Accounts and Microsoft Azure logins, send us email.
  • Welcome to contribute to the project (on GitHub).
backup
Main Page

MediaWiki has been successfully installed.

Consult the User's Guide for information on using the wiki software.

Getting started [edit]

- Configuration settings list
- MediaWiki FAQ
- MediaWiki release mailing list
- Localise MediaWiki for your language

This page was last modified on 17 February 2016, at 20:33.