FINDING INFORMATION FLOW BUGS WITH SYMBOLIC EXECUTION

JOSÉ FRAGOSO SANTOS
ASSISTANT PROFESSOR, DEI

NOVEMBER 2019 @ SOFTWARE SECURITY (MEIC)
Goal: Understand how to use self-composition to find information flow bugs

1. Motivation: Information flow bugs on the Web

2. Program Properties and Non-interference

3. Self-Composition + Symbolic Execution

4. Concolic Symbolic Execution
RESEARCH PROGRAM — VERIFICATION AND TESTING

Verification and Testing of JavaScript (Web) Programs

- **APLAS’16**
  - **DOM SSL Logic**: Axiomatic Specification of DOM API

- **CADE’17**
  - **JS Specs**: Separation Logic Specifications for JS

- **PPDP’18**
  - **Cosette**: Symbolic Debugging for JS

- **POPL’18**
  - **JaVerT**: JavaScript Verification Toolchain

- **POPL’19**
  - **JaVerT 2.0**: JavaScript Verification and Testing
Research Program – Verification and Testing

Verification and Testing of JavaScript (Web) Programs

Winner of the Facebook Continuous Reasoning Research Award (50K USD)
Research Program – Verification and Testing

Verification and Testing of JavaScript (Web) Programs

Collaboration with Amazon to verify critical components of the AWS Encryption SDK
Information Flow Control for Web Programs

- Inlining-Compiler for JS Information Flow Control (SEC’14)
- Modular Monitor Extensions for JS APIs (TGC’15)
- Information Flow Monitor for the DOM API (TGC’14)
- Hybrid Typing Secure Information Flow in JS (TGC’15)
- Mashic – Automatic Sandboxing of third party untrusted code (Journal of Computer Security, 16)
I. INFORMATION FLOW CONTROL ON THE WEB
Browser extensions are often implemented directly in JavaScript

- Browser extensions execute with elevated privileges
- Web apps can communicate with the extensions executing in the browser via the Browser API
- Malicious Web Apps can exploit browser extension leaks to obtain user-sensitive data

EmPoWeb* found often than 100 leaks in JS browser extensions

*EmPoWeb: Empowering Web Applications with Browser Extensions, Dolière Francis Somé, 2019
EmPoWeb* found often than 100 leaks in JS browser extensions

It uses a simple syntactic analysis. It explicitly looks for expressions with a fixed form:

```
chrome.cookies.getAll
chrome.cookies.remove
```

**Simple syntactic analysis are not enough!**

But there are multiple other ways to access these values that would not be caught by this analysis.
Obfuscated JS Leak

Leaking 1 bit

```javascript
function getBit(x) {
    var a = [ "a", "b", "c" ];
    if ((x % 2) === 0) {
        Object.defineProperty(a, "1", {value: "d", configurable: false} )
    }
    a.length = 1;
    return (a.length === 1) ? 1 : 0;
}
```

Implicit Information flow via a property descriptor
Obfuscated JS Leak

Leaking 1 bit

```javascript
function getBit(x) {
    var op = {};
    var f = function () {};
    f.prototype = op;
    var o = new f();
    if ((x % 2) === 0) {
        Object.defineProperty(op, "foo", {value: 0, configurable: false})
    }
    o.foo = 1;
    return o.foo;
}
```

Implicit Information flow via **prototype inheritance**
OBFUSCATED JS LEAK

Bit by bit, we can leak all bits

```javascript
function getBits(x) {
    var bits = [];
    while (x > 0) {
        bits.push(getBit(x));
        x = x >> 1;
    }
    return (bits.length === 0) ? 0 : bits.reverse();
}
```
And now we can learn:

```javascript
var x = chrome;
var y1 = "cook";
var y2 = "ies";
var z1 = "rem";
var z2 = "ove";
var w = getBits(x[y1+y2][z1+z2])
```
Take-home message

Malicious code can exploit corner case behaviors of the JavaScript semantics to encode sophisticated information flow

We need to do better than a simple syntactic analysis!
II. PROGRAM PROPERTIES
Program property = set of “behaviors”

Verification: verifying a given program $P$ with respect to a given property $S$ means proving that all the behaviors of $P$ are contained in $S$

$[P] \subseteq S$
**Verification vs Bug-finding**

Program property = set of “behaviors”

Bug Finding: debugging a program $P$ with respect to a given property $S$ means finding a behavior of $P$ that is not in $S$

$$[P] \cap \overline{S} \neq \emptyset$$
Verification vs Bug-finding

**Verification**

\[ [P] \subseteq S \]

Hard: *for all*

**Bug-finding**

\[ [P] \cap \overline{S} \neq \emptyset \]

Easy: *exists*
**PROGRAM PROPERTIES**

**Question:** How do we define the set of behaviors?

$[P]$?

Depending on how we define the set of allowed behaviors, we get different classes of properties.
Safety Properties: Nothing **bad** ever happens

Liveness Properties: Something **good** eventually happens
**TRACE PROPERTIES**

**Safety Properties:** Nothing **bad** ever happens  
- Type Safety  
- Memory Safety (no null pointer exceptions)

**Liveness Properties:** Something **good** eventually happens
Trace Properties

Safety Properties: Nothing bad ever happens
  • Type Safety
  • Memory Safety (no null pointer exceptions)

Liveness Properties: Something good eventually happens
  • Termination
  • Absence of memory leaks
Trace Properties – Formally

Program property = set of “behaviors”

How do we formally define program traces?

Trace property = set of program traces
Syntax of While

\[ e_1, e_2 \in \mathcal{E} \triangleq n \mid x \mid \Theta e_1 \mid e_1 \oplus e_2 \]

\[ s_1, s_2 \in \mathcal{S} \triangleq \text{skip} \mid x := e \mid s_1; s_2 \]
\[ \mid \text{if (e)}\{ s_1 \}\text{else} \{ s_2 \} \]
\[ \mid \text{while (e)}\{ s_1 \} \]
**Operational Semantics - While Language**

**Syntax of While**

\[ e_1, e_2 \in E \triangleq n \mid x \mid \Theta e_1 \mid e_1 \oplus e_2 \]

\[ s_1, s_2 \in S \triangleq \text{skip} \mid x := e \mid s_1 ; s_2 \]

\[ \mid \text{if } (e) \{ s_1 \} \text{else } \{ s_2 \} \]

\[ \mid \text{while } (e) \{ s_1 \} \]

**Small-step Transition**

\[ \langle \rho, s \rangle \rightarrow \langle \rho', s' \rangle \]

**State = Variable Store**

\[ \rho \in \text{Store} : \text{PVar} \rightarrow \mathbb{N} \]
A simple **While Language - Semantics**

**Assignment**

\[
\rho' = \rho \left[ x \mapsto [e]_{\rho} \right]
\]

\[\langle \rho, x := e \rangle \rightarrow \langle \rho', \text{skip} \rangle\]

**If - True**

\[\boxed{[e]_{\rho} \neq 0}\]

\[\langle \rho, \text{if}(e)\{ \ s_1 \} \text{else} \{ \ s_2 \} \rangle \rightarrow \langle \rho, s_1 \rangle\]

**If - False**

\[\boxed{[e]_{\rho} = 0}\]

\[\langle \rho, \text{if}(e)\{ \ s_1 \} \text{else} \{ \ s_2 \} \rangle \rightarrow \langle \rho, s_2 \rangle\]

**Seq - 1**

\[\langle \rho, s_1 \rangle \rightarrow \langle \rho, s'_1 \rangle\]

\[\langle \rho, s_1; s_2 \rangle \rightarrow \langle \rho, s'_1; s_2 \rangle\]

**Seq - 2**

\[\langle \rho, \text{skip}; s_2 \rangle \rightarrow \langle \rho, s_2 \rangle\]

**While**

\[s' = \text{if}(e)\{ \ s; \text{while} (e)\{s\} \} \text{else} \{ \ \text{skip} \ \}\]

\[\langle \rho, \text{while} (e)\{s\} \rangle \rightarrow \langle \rho, s' \rangle\]
A simple While Language - Semantics

Division by 0 generates

\[
\begin{align*}
\text{Assignment - Error} & \quad [e]_\rho = \bot \\
\frac{}{\langle \rho, x := e \rangle \rightarrow \langle \rho, \text{skip} \rangle} \\
\text{If - Error} & \quad [e]_\rho = \bot \\
\frac{}{\langle \rho, \text{if}\ (e)\{\ s_1\}\text{else}\{\ s_2\}\rangle \rightarrow \langle \rho, \text{skip} \rangle}
\end{align*}
\]
Trace properties - Formally

Trace property = set of program traces

Program Trace = ?

\[[s] \triangleq \{[\langle \rho_0, s_0 \rangle, ..., \langle \rho_n, s_n \rangle] | s_0 = s \land s_n = \text{skip} \land \forall 0 \leq i < n \langle \rho_i, s_i \rangle \rightarrow \langle \rho_{i+1}, s_{i+1} \rangle\}\]
trace properties - formally

trace property = No division by 0

\[ \text{NoDivZero} \triangleq \{[[\rho_0, s_0], ..., [\rho_n, s_n]] \mid s_n = \text{skip} \land \rho_n \neq \frac{1}{0} \land \forall 0 \leq i < n \rho_i, s_i \rightarrow [\rho_{i+1}, s_{i+1}] \} \]
Trace properties - formally

Trace property = Termination

Termination $\triangleq \{ [\langle \rho_0, s_0 \rangle, \ldots, \langle \rho_n, s_n \rangle] \mid s_n = \text{skip} \land \forall 0 \leq i < n \langle \rho_i, s_i \rangle \rightarrow \langle \rho_{i+1}, s_{i+1} \rangle \}$
Trace Properties – Formally

$T_{x0} = \text{Programs that terminate with } x \text{ set to 0}$

$T_{0} = \text{Programs that terminate with a all variables set to 0}$
**Trace Properties - Summary**

- Trace Properties
  - Safety Properties
    - State Properties
  - Liveness Properties
WHAT ABOUT NON-INTERFERENCE?

\[ \mathcal{N} \mathcal{I}(\Gamma) \triangleq \{ s \mid \forall \rho_1, \rho_2 . \rho_1 =^L \rho_2 \land \langle \rho_1, s \rangle \rightarrow^* \langle \rho'_1, \text{skip} \rangle \land \langle \rho_2, s \rangle \rightarrow^* \langle \rho'_2, \text{skip} \rangle \implies \rho'_1 =^L \rho'_2 \} \]
Non-Interference is a $2$-trace property

\[
NI(\Gamma) \triangleq \{ s \mid \forall \rho_1, \rho_2, \rho_1 =^\Gamma \rho_2 \land \langle \rho_1, s \rangle \rightarrow^* \langle \rho_1', \text{skip} \rangle \\
\land \langle \rho_2, s \rangle \rightarrow^* \langle \rho_2', \text{skip} \rangle \implies \rho_1' =^\Gamma \rho_2' \}
\]

\[
NI(\Gamma) \triangleq \{ ([\langle \rho_0, s_0 \rangle, ..., \langle \rho_n, s_n \rangle], [\langle \rho'_0, s'_0 \rangle, ..., \langle \rho'_m, s'_m \rangle]) \\
\mid \forall 0 \leq i < n \langle \rho_i, s_i \rangle \rightarrow \langle \rho_{i+1}, s_{i+1} \rangle \\
\land \forall 0 \leq i < m \langle \rho'_i, s'_i \rangle \rightarrow \langle \rho'_{i+1}, s'_{i+1} \rangle \\
\land s_0 = s'_0 \land (\rho_0 =^\Gamma \rho'_0 \implies \rho_n =^\Gamma \rho'_m) \}
\]
\( \mathcal{NI}(\Gamma) \triangleq \{ s | \forall \rho_1, \rho_2 . \rho_1 =^L \rho_2 \land \langle \rho_1, s \rangle \rightarrow^* \langle \rho'_1, \text{skip} \rangle \land \langle \rho_2, s \rangle \rightarrow^* \langle \rho'_2, \text{skip} \rangle \implies \rho'_1 =^L \rho'_2 \} \)

A pair of stores \((\rho_1, \rho_2)\) that prove that:

\[ s \notin \mathcal{NI}(\Gamma) \]

\[ \langle \rho_1, s \rangle \rightarrow^* \langle \rho'_1, \text{skip} \rangle \land \langle \rho_2, s \rangle \rightarrow^* \langle \rho'_2, \text{skip} \rangle \land \rho_1 =^L \rho_2 \land \rho'_1 \neq ^L \rho'_2 \]
2-Trace Properties: Properties of 2 traces
  • Non-Interference
  • Dependency

N-Trace Properties: Meta-dependencies
PROGRAM PROPERTIES - SUMMARY

Hyper Properties

Trace Properties

Safety Properties

State Properties

Liveness Properties
II. SELF-COMPOSITION + SYMBOLIC EXECUTION
SELF-COMPOSITION — THE MAIN IDEA

Idea: Reduce non-interference to a safety property by transpiling the given program

\[ s \in \mathcal{NI}(\Gamma) \iff \left[ C(s) \right] \subseteq \mathcal{T}(\Gamma) \]

\( \mathcal{T}(\Gamma) \) - a safety property that only depends on \( \Gamma \)

\( C \) - a transpiler that computes the self-composition of \( s \)
**SELF-COMPOSITION — WHY?**

**Idea:** Reduce non-interference to a safety property by transpiling the given program.

**SCALABLE PROGRAM ANALYSES ARE HARD TO DESIGN AND IMPLEMENT, ESPECIALLY WHEN TARGETING REAL-WORLD LANGUAGES**

**Why?** We can use an existing analysis to check the safety property instead of building a new analysis from scratch to check non-interference.

**WE ARE GOING TO USE SYMBOLIC EXECUTION**
SELF-COMPOSITION — EXAMPLE 1

\[
\begin{align*}
&\text{assume}(l_1 = l_2); \\
&\text{assert}(l_1 = l_2)
\end{align*}
\]

WE DON’T HAVE TO KNOW! WE CAN USE SYMBOLIC EXECUTION...

Does the assertion hold?
\[ l := h \]

We are going to execute the generated program symbolically

\[
\text{assume}(l1 = l2); \\
l1 := h1; \\
l2 := h2; \\
\text{assert}(l1 = l2)
\]

Instead of using concrete values, we use **symbolic variables**
SELF-COMPOSITION — EXAMPLE 1

```plaintext
l := h
assume(l1 = l2);
l1 := h1;
l2 := h2;
assert(l1 = l2)
```

True, [ l1→#l1, h1→#h1, l2→#l2, h2→#h2]

Symbolic Store

Path Condition: conjunction of all the expressions on which the execution has branched before reaching the current execution point
SELF-COMPOSITION — EXAMPLE 1

\[ l := h \]

\[ \text{assume}(l_1 = l_2); \]
\[ l_1 := h_1; \]
\[ l_2 := h_2; \]
\[ \text{assert}(l_1 = l_2) \]

True, [ \[ l_1 \rightarrow \#l_1, h_1 \rightarrow \#h_1, l_2 \rightarrow \#l_2, h_2 \rightarrow \#h_2 \] ]

\[ \text{assume}(l_1 = l_2) \]
\[ \#l_1 = \#l_2, [ \[ l_1 \rightarrow \#l_1, h_1 \rightarrow \#h_1, l_2 \rightarrow \#l_2, h_2 \rightarrow \#h_2 \] ] \]

\[ l_1 := h_1 \]
\[ \#l_1 = \#l_2, [ \[ l_1 \rightarrow \#h_1, h_1 \rightarrow \#h_1, l_2 \rightarrow \#l_2, h_2 \rightarrow \#h_2 \] ] \]

\[ l_2 := h_2 \]
\[ \#l_1 = \#l_2, [ \[ l_1 \rightarrow \#h_1, h_1 \rightarrow \#h_1, l_2 \rightarrow \#h_2, h_2 \rightarrow \#h_2 \] ] \]
SELF-COMPOSITION — EXAMPLE 1

\[ l := h \]

\[ \text{assume}(l1 = l2); \]
\[ l1 := h1; \]
\[ l2 := h2; \]
\[ \text{assert}(l1 = l2) \]

\[ l2 := h2 \]
\[ \#l1=\#l2, \ [ l1\rightarrow\#h1, h1\rightarrow\#h1, l2\rightarrow\#h2, h2\rightarrow\#h2] \]

\[ \text{assert}(l1 = l2) \]

\[ \not\checkmark \]

\[ (#l1 = #l2) \Rightarrow (#h1 = #h2) \quad \text{Valid?} \]

\[ (#l1 = #l2) \land (#h1 \neq #h2) \quad \text{SAT?} \]
SELF-COMPOSITION – EXAMPLE 1

\[ l := h \]

\[ \text{assume}(l_1 = l_2); \]
\[ l_1 := h_1; \]
\[ l_2 := h_2; \]
\[ \text{assert}(l_1 = l_2) \]

\[ l_2 := h_2 \]

\[ \#l_1 = \#l_2, [ l_1\rightarrow\#h_1, h_1\rightarrow\#h_1, l_2\rightarrow\#h_2, h_2\rightarrow\#h_2] \]
\[ \text{assert}(l_1 = l_2) \]

\[ (#l_1 = #l_2) \land (#h_1 \neq #h_2) \] SAT?

YES! [ #l_1\rightarrow0, #h_1\rightarrow0, #l_2\rightarrow0, #h_2\rightarrow1]
if(h) {
  l := 1
} else {
  skip
}

assume(l1 = l2);
if (h1) { l1 := 1 } else { skip };
if (h2) { l2 := 1 } else { skip };
assert(l1 = l2)
\textbf{SELF-COMPOSITION — EXAMPLE 2}

\texttt{assume(l1 = l2);}
\texttt{if (h1) {
  l1 := 1
} else { skip };}
\texttt{if (h2) {
  l2 := 1
} else { skip };}
\texttt{assert(l1 = l2)}

\text{True, [ l1→#l1, h1→#h1, l2→#l2, h2→#h2]}

\text{assume(l1 = l2)}

\text{#l1=#l2, [ l1→#l1, h1→#h1, l2→#l2, h2→#h2]}

\text{if (h1)}

\text{#h1 = 0}

\text{#h1 ≠ 0}

\textbf{Next Slide}
**Self-Composition — Example 2**

```plaintext
assume(l1 = l2);
if (h1) {
    l1 := 1
} else {
    skip
}
if (h2) {
    l2 := 1
} else {
    skip
}
assert(l1 = l2)
```

```plaintext
#h1 = 0 
if (h1) 
... 
#h1 ≠ 0

#l1=#l2 ∧ #h1 ≠ 0, 
[ l1->#l1, h1->#h1, l2->#l2, h2->#h2] 

l1 := 1 

#l1=#l2 ∧ #h1 ≠ 0, 
[ l1->1, h1->#h1, l2->#l2, h2->#h2] 
```
assume(\(l_1 = l_2\));
if (h1) {
  \(l_1 := 1\)
} else { skip }
if (h2) {
  \(l_2 := 1\)
} else { skip }
assert(\(l_1 = l_2\))
SELF-COMPOSITION — EXAMPLE 2

```plaintext
assume(l1 = l2);
if (h1) {
l1 := 1
} else { skip }
if (h2) {
l2 := 1
} else { skip }
assert(l1 = l2)
```

#h2 = 0

if (h2)

#h2 ≠ 0

#l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 ≠ 0,
[ l1→1, h1→#h1, l2→#l2, h2→#h2]

l2 := 1

#l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 ≠ 0,
[ l1→1, h1→#h1, l2→1, h2→#h2]
assume(l1 = l2);
if (h1) {
  l1 := 1
} else { skip }
if (h2) {
  l2 := 1
} else { skip }
assert(l1 = l2)

l2 := 1

SAT?

NO! No bug found
assume(l1 = l2);
if (h1) {
    l1 := 1
} else {
    skip
};
if (h2) {
    l2 := 1
} else {
    skip
};
assert(l1 = l2)

#h2 ≠ 0
... if (h2)
  #h2 = 0
  #l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 = 0,
  [ l1→1, h1→#h1, l2→#l2, h2→#h2]
  skip
  #l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 = 0,
  [ l1→1, h1→#h1, l2→#l2, h2→#h2]
**SELF-COMPOSITION — EXAMPLE 2**

```plaintext
assume(l1 = l2);
if (h1) {
    l1 := 1
} else { skip }
if (h2) {
    l2 := 1
} else { skip }
assert(l1 = l2)
```

\[\#l1 = \#l2 \land \#h1 \neq 0 \land \#h2 = 0,\]
\[\text{[ } l1\rightarrow 1, h1\rightarrow \#h1, l2\rightarrow \#l2, h2\rightarrow \#h2\text{]}\]

\[\text{assert}(l1 = l2)\]
\[\#l1 = \#l2 \land \#h1 \neq 0 \land \#h2 = 0 \land 1 \neq \#l2\text{ SAT?}\]

\[\text{YES! [ } \#l1\rightarrow 0, \#h1\rightarrow 1, \#l2\rightarrow 0, \#h2\rightarrow 0 \text{ ]}\]
SELF-COMPOSITION — EXAMPLE 2

True, \([ l1\rightarrow#l1, h1\rightarrow#h1, l2\rightarrow#l2, h2\rightarrow#h2]\)

- \#h1 = 0
- \#h1 \neq 0

- \#h2 = 0
- \#h2 \neq 0

- \#h1 = 0 \land \#h2 = 0
- \#h1 = 0 \land \#h2 \neq 0
- \#h1 \neq 0 \land \#h2 = 0
- \#h1 \neq 0 \land \#h2 \neq 0
If we find a bug, we know that the program is not secure

But what if we do not find a bug?

Is the program secure?
**Self-Composition — Example 2**

But what if we do not find a bug?

The program is secure if we covered all the possible execution paths.

How can we know that?
How do we know if we covered all possible execution paths?

The disjunction of all final path conditions must be True.

#h1 = 0 \land #h2 = 0 \lor #h1 = 0 \land #h2 \neq 0 \lor #h1 \neq 0 \land #h2 = 0 \lor #h1 \neq 0 \land #h2 \neq 0 = True
z := 1;
if(h){ x := 1 }  
   else { skip };
if(!h) { x := z }  
   else { skip };
l := x + y

assume(l1=l2 and z1=z2 and y1=y2)
z1 := 1;
if(h1){ x1 := 1 }  
   else { skip };
if(!h1) { x1 := z1 }  
   else { skip };
l1 := x1 + y1;
z2 := 1;
if(h2){ x2 := 1 }  
   else { skip };
if(!h2) { x2 := z2 }  
   else { skip };
l2 := x2 + y2;
assert(l1=l2 and z1=z2 and y1=y2)
The final symbolic store is always the same!

What is the SAT query?

\[ [ l_1 \rightarrow 1 + \#y_1, l_2 \rightarrow 1 + \#y_2, h_1 \rightarrow \#h_1, h_2 \rightarrow \#h_2, y_1 \rightarrow \#y_1, y_2 \rightarrow \#y_2, z_1 \rightarrow \#z_1, z_2 \rightarrow \#z_2 ] \]
Self-Composition — Formally

Idea: Reduce non-interference to a safety property by transpiling the given program

\[ s \in \mathcal{NI}(\Gamma) \iff \mathcal{C}(s) \subseteq \mathcal{T}(\Gamma) \]

- \( \mathcal{T}(\Gamma) \) - a safety property that only depends on \( \Gamma \)
- \( \mathcal{C} \) - a transpiler that computes the self-composition of \( s \)
SELF-COMPOSITION — FORMALLY

\[ \text{dom}(\theta_1) = \text{dom}(\theta_2) = \text{vars}(s) \quad \text{rng}(\theta_1) \cap \text{rng}(\theta_2) = \emptyset \]

\[ C(s) \overset{\Delta}{=} \theta_1(s); \theta_2(s) \]

\[ \mathcal{T}(\Gamma) \overset{\Delta}{=} \{ [[\rho_0, s_0], \ldots, [\rho_n, s_n]] \mid s_n = \text{skip} \land \forall_{0 \leq i < n} \langle \rho_i, s_i \rangle \rightarrow \langle \rho_{i+1}, s_{i+1} \rangle \land (\land\{ (\rho_0(\theta_1(x)) = \rho_0(\theta_2(x))) \mid \Gamma(x) = L \} \Rightarrow \land\{ (\rho_n(\theta_1(x)) = \rho_n(\theta_2(x))) \mid \Gamma(x) = L \}) \} \]
\[
\text{SELF-COMPOSITION — FORMALLY}
\]

\[
\begin{align*}
\text{dom}(\theta_1) &= \text{dom}(\theta_2) = \text{vars}(s) & \text{rng}(\theta_1) \cap \text{rng}(\theta_2) &= \emptyset \\
\text{s}_{\text{assume}} &= \text{assume} \left( \bigwedge \{ (\theta_1(x) = \theta_2(x)) \mid \Gamma(x) = L \} \right) \\
\text{s}_{\text{assert}} &= \text{assert} \left( \bigwedge \{ (\theta_1(x) = \theta_2(x)) \mid \Gamma(x) = L \} \right) \\
\end{align*}
\]

\[
C(s) \triangleq \text{s}_{\text{assume}}; \theta_1(s); \theta_2(s); \text{s}_{\text{assert}}
\]
IV. CONCOLIC SYMBOLIC EXECUTION
CONCOLIC SYMBOLIC EXECUTION — THE MAIN IDEA

Idea: Execute the program concretely and symbolically at the same time

Why?
- Symbolic execution is often too expensive...
- Back-end constraint solvers sometimes (often!) cannot find the answer: UNKNOWN

P. Godefroid
Concolic Testing: Main Algorithm

Input\(_0\) = *Pick random vector*
Coverage = *False*
i = 0

*While* (Input\(_i\) ≠ *NULL*) {
  (RES\(_i\), PC\(_i\)) = Run Program with Input\(_i\)
  Coverage = Coverage ∨ PC\(_i\)
  Input\(_{i+1}\) ∈ *Models*(¬Coverage)
  i = i + 1
}

Concolic Symbolic Execution — The Main Idea
Concolic Symbolic Execution — the main idea

\[ z := 2\cdot y; \]
\[ \text{if } (z = x) \{ \]
\[ \quad \text{if } (x > y+10) \{ \]
\[ \quad \quad \text{assert(false)} \]
\[ \quad \} \text{ else } \{ \text{ skip } \}; \]
\[ \} \text{ else } \{ \text{ skip } \} \]

**Step 1:**

Inputs_0 = [ x\rightarrow 22, y\rightarrow 7 ]
(RES_1, PC_1) = (OK, (x \neq 2\cdot y))
Coverage = (x \neq 2\cdot y)
Inputs_1 = [x\rightarrow 2, y\rightarrow 1 ]
**Concolic Symbolic Execution — The Main Idea**

\[
z := 2*y;
\]
\[
\text{if } (z = x) \{
\text{if } (x > y+10) \{
\text{assert(false)}
\} \text{ else } \{ \text{skip } \};
\} \text{ else } \{ \text{skip } \}
\]

**Step 2:**

Inputs\(_1\) = [x\!\!2,y\!\!1 ]

(RES\(_2\), PC\(_2\)) = (OK,(x =2*y)∧(x≤ y+10))

Coverage = (x ≠ 2*y) ∨

\((x = 2*y)∧(x≤ y+10))

Inputs\(_2\) = [x\!\!30,y\!\!15 ]
CONCOLIC SYMBOLIC EXECUTION — THE MAIN IDEA

\[ z := 2\times y; \]
\[ \text{if } (z = x) \{ \]
\[ \quad \text{if } (x > y+10) \{ \]
\[ \quad \quad \text{assert(false)} \]
\[ \quad \} \text{ else } \{ \text{ skip } \}; \]
\[ \} \text{ else } \{ \text{ skip } \} \]

**Step 3:**

Inputs\(_2\) = [x\to 30, y\to 15]

(RES\(_3\), PC\(_3\)) = (FAIL, (x = 2\times y) \land (x>y+10))

Coverage = (x \neq 2\times y) \lor
\[(x = 2\times y) \land (x<=y+10)) \lor
\[(x = 2\times y) \land (x>y+10))\]

Inputs\(_2\) = NULL
A LOT MORE TO COVER...

1. Symbolic execution with **data structures**
   - Lazy-Initialization
   - Data-structure **unfolding**

2. Declassification

3. Other Security Properties: **Confinement**

4. Invariants and verification
Bounded model checking for TypeScript via symbolic execution and compilation

Code-stepping regular expressions in the browser

Building a symbolic execution engine for your favorite programming language
Bounded model checking for JavaScript regular expressions

Symbolically debugging secure information flow in the browser

And more... Check my website:
http://web.ist.utl.pt/jose.fragoso
MASTER PROJECTS

Information Flow Control
Web Programs
Symbolic Execution

Potential collaboratons with:

Imperial College
London

Ínria

IBM Watson