Finding Information Flow bugs with Symbolic Execution

José Fragoso Santos

Assistant Professor, DEI

November 2019 @ Software Security (MEIC)

CONTENTS

GOAL: UNDERSTAND HOW TO USE SELF-COMPOSITION TO FIND INFORMATION FLOW BUGS

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

APLAS'16

DOM ...
Spe...

CADE'17

...ogic
...S

PPDP'18

Cosette: Symbolic Execution for JS

...'19

JaVerT 2.0: Compositional Symbolic Execution for JS

Winner of the Facebook Continuous Reasoning Research Award (50K USD)

facebook research

Research Program – Verification and Testing

Verification and Testing of JavaScript (Web) Programs

APLAS'16

DOM ... Spec...

CADE'17

...ogic

PPDP'18

Cosette: Symbolic Execution for JS

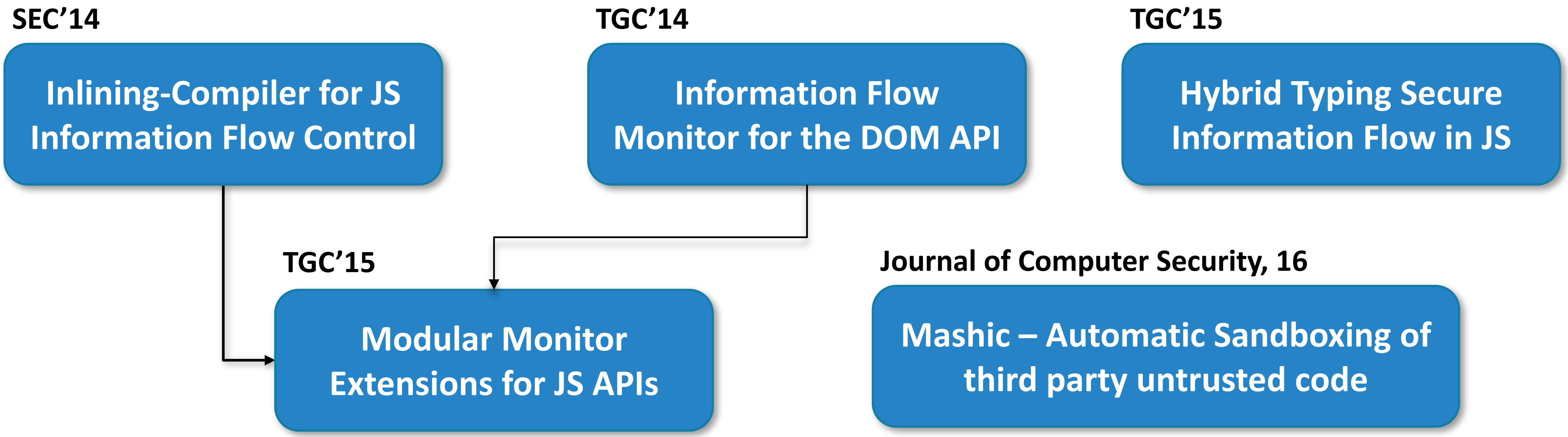
...'19

JaVerT 2.0: Compositional Symbolic Execution for JS

Collaboration with Amazon to verify critical components of the AWS Encryption SDK

RESEARCH PROGRAM – INFORMATION FLOW CONTROL

Information Flow Control for Web Programs



I. Information Flow Control on the Web

JS Security Vulnerabilities – Browser Extensions

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

JS Security Vulnerabilities – Browser Extensions

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:

Simple syntactic analysis are not enough!

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

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

Obfuscated JS Leak

Leaking 1 bit

```
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

```
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

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

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

Obfuscated JS Leak

And now we can learn:

`chrome.cookies.remove`

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

Obfuscated JS Leak

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

Verification vs Bug-finding

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*

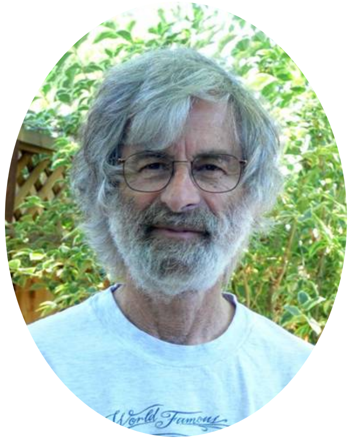
PROGRAM PROPERTIES

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.

TRACE PROPERTIES



Leslie Lamport

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?

Traces

Trace property = set of program traces

Operational Semantics - While Language

Syntax of While

$$e_1, e_2 \in \mathcal{E} \triangleq n \mid \mathsf{x} \mid \ominus e_1 \mid e_1 \oplus e_2$$

$$\begin{aligned} s_1, s_2 \in \mathcal{S} \triangleq\ & \mathsf{skip} \mid \mathsf{x} := e \mid s_1; s_2 \\ & \mid \mathsf{if}\,(e)\{\, s_1 \}\mathsf{else}\,\{\, s_2 \,\} \\ & \mid \mathsf{while}\,(e)\{s_1\} \end{aligned}$$

Small-Step Operational Semantics

Gordon Plotkin

OPERATIONAL SEMANTICS - WHILE LANGUAGE

Syntax of While

$$e_1, e_2 \in \mathcal{E} \triangleq n \mid \mathsf{x} \mid \ominus e_1 \mid e_1 \oplus e_2$$

$$\begin{aligned} s_1, s_2 \in \mathcal{S} \triangleq\ & \mathsf{skip} \mid \mathsf{x} := e \mid s_1; s_2 \\ & \mid \mathsf{if}\,(e)\{\ s_1\}\mathsf{else}\ \{\ s_2\ \} \\ & \mid \mathsf{while}\,(e)\{s_1\} \end{aligned}$$

Small-step Transition

$$\langle \rho, s \rangle \rightarrow \langle \rho', s' \rangle$$

State = Variable Store

$$\rho \in \mathsf{Store} : \mathsf{PVar} \rightharpoonup \mathbb{N}$$

A simple While Language - Semantics

Assignment

$$\frac{\rho' = \rho\left[x \mapsto [\![e]\!]_\rho\right]}{\langle \rho, \mathsf{x} := e\rangle \longrightarrow \langle \rho', \mathsf{skip}\,\rangle}$$

If - True

$$\frac{[\![e]\!]_\rho \neq 0}{\langle \rho, \mathsf{if}\,(e)\{\ s_1\}\mathsf{else}\ \{\ s_2\ \}\rangle \longrightarrow \langle \rho, s_1\rangle}$$

If - False

$$\frac{[\![e]\!]_\rho = 0}{\langle \rho, \mathsf{if}\,(e)\{\ s_1\}\mathsf{else}\ \{\ s_2\ \}\rangle \longrightarrow \langle \rho, s_2\rangle}$$

Seq - 1

$$\frac{\langle \rho, s_1\rangle \longrightarrow \langle \rho, s_1'\rangle}{\langle \rho, s_1; s_2\rangle \longrightarrow \langle \rho, s_1'; s_2\rangle}$$

Seq - 2

$$\langle \rho, \mathsf{skip}\,; s_2\rangle \longrightarrow \langle \rho, s_2\rangle$$

While

$$\frac{s' = \mathsf{if}\,(e)\{\ s; \mathsf{while}\,(e)\{s\}\}\mathsf{else}\ \{\ \mathsf{skip}\ \ \}}{\langle \rho, \mathsf{while}\,(e)\{s\}\rangle \longrightarrow \langle \rho, s'\rangle}$$

A simple While Language - Semantics

Division by 0 generates $\lightning$

$$\textsc{Assignment - Error}\quad \frac{[\![e]\!]_\rho = \lightning}{\langle \rho, \mathsf{x} := e \rangle \rightarrow \langle \lightning, \mathsf{skip} \rangle}$$

$$\textsc{If - Error}\quad \frac{[\![e]\!]_\rho = \lightning}{\langle \rho, \mathsf{if}\,(e)\{\, s_1 \}\mathsf{else}\,\{\, s_2 \,\}\rangle \rightarrow \langle \lightning, \mathsf{skip} \rangle}$$

TRACE PROPERTIES - FORMALLY

Trace property = set of program traces

Program Trace = ?

$$[\![s]\!] \triangleq \{[\langle \rho_0, s_0 \rangle, \ldots, \langle \rho_n, s_n \rangle] \mid s_0 = s \wedge s_n = \text{skip} \wedge \forall_{0 \leq i < n} \langle \rho_i, s_i \rangle \rightarrow \langle \rho_{i+1}, s_{i+1} \rangle\}$$

TRACE PROPERTIES - FORMALLY

Trace property = No division by 0

$$\texttt{NoDivZero} \triangleq \{[\langle \rho_0, s_0\rangle, \ldots, \langle \rho_n, s_n\rangle] \mid s_n = \mathsf{skip} \;\wedge\; \rho_n \neq \lightning \\ \wedge\, \forall_{0 \le i < n} \langle \rho_i, s_i\rangle \longrightarrow \langle \rho_{i+1}, s_{i+1}\rangle\}$$

TRACE PROPERTIES - FORMALLY

Trace property = Termination

$$\texttt{Termination} \triangleq \{[\langle \rho_0, s_0\rangle, \ldots, \langle \rho_n, s_n\rangle] \mid s_n = \mathsf{skip} \\ \wedge \forall_{0 \le i < n} \langle \rho_i, s_i\rangle \longrightarrow \langle \rho_{i+1}, s_{i+1}\rangle\}$$

Trace Properties - Formally

Tx0 = Programs that terminate with **x** set to **0**

T0 = Programs that terminate with a **all variables** set to **0**

Trace Properties - Summary

What about non-interference?

J. Meseguer

J. Goguen

Security Labeling

$$\Gamma : \mathsf{PVar} \rightharpoonup \mathcal{L}$$

Security Labeling

$$\mathcal{L} = \langle \{L, H\}, \sqsubseteq \rangle$$

$$\begin{aligned}\mathcal{NI}(\Gamma) \triangleq \{s \mid \forall \rho_1, \rho_2 \,.\, \rho_1 =^{\Gamma}_{L} \rho_2 \;\wedge\; \langle \rho_1, s\rangle \rightarrow^* \langle \rho_1', \mathsf{skip}\,\rangle \\ \wedge\; \langle \rho_2, s\rangle \rightarrow^* \langle \rho_2', \mathsf{skip}\,\rangle \implies \rho_1' =^{\Gamma}_{L} \rho_2'\}\end{aligned}$$

NON-INTERFERENCE: 2-TRACE PROPERTY

Non-Interference is a **2-trace** property

$$\mathcal{NI}(\Gamma) \triangleq \{s \mid \forall \rho_1, \rho_2 \,.\, \rho_1 =_L^\Gamma \rho_2 \ \wedge\ \langle \rho_1, s\rangle \rightarrow^* \langle \rho_1', \text{skip}\,\rangle \\ \wedge\, \langle \rho_2, s\rangle \rightarrow^* \langle \rho_2', \text{skip}\,\rangle \implies \rho_1' =_L^\Gamma \rho_2'\}$$

$$\mathcal{NI}(\Gamma) \triangleq \{([\langle \rho_0, s_0\rangle, \ldots, \langle \rho_n, s_n\rangle], [\langle \rho_0', s_0'\rangle, \ldots, \langle \rho_m', s_m'\rangle]) \\ \mid \forall_{0 \leq i < n} \langle \rho_i, s_i\rangle \rightarrow \langle \rho_{i+1}, s_{i+1}\rangle \\ \wedge\, \forall_{0 \leq i < m} \langle \rho_i', s_i'\rangle \rightarrow \langle \rho_{i+1}', s_{i+1}'\rangle \\ \wedge\, s_0 = s_0' \ \wedge\ (\rho_0 =_L^\Gamma \rho_0' \implies \rho_n =_L^\Gamma \rho_m')\}$$

Information flow Bug

$$\mathcal{NI}(\Gamma) \triangleq \{s \mid \forall \rho_1, \rho_2 \,.\, \rho_1 =_L^\Gamma \rho_2 \;\wedge\; \langle \rho_1, s\rangle \rightarrow^* \langle \rho_1', \mathsf{skip}\,\rangle \\ \wedge\, \langle \rho_2, s\rangle \rightarrow^* \langle \rho_2', \mathsf{skip}\,\rangle \implies \rho_1' =_L^\Gamma \rho_2'\}$$

A pair of stores (ρ_1, ρ_2) that prove that:

$$s \notin \mathcal{NI}(\Gamma)$$

$$\langle \rho_1, s\rangle \rightarrow^* \langle \rho_1', \mathsf{skip}\,\rangle \;\wedge\; \langle \rho_2, s\rangle \rightarrow^* \langle \rho_2', \mathsf{skip}\,\rangle \\ \wedge\; \rho_1 =_L^\Gamma \rho_2 \;\wedge\; \rho_1' \neq_L^\Gamma \rho_2'$$

HYPER-PROPERTIES

F. Schneider

2-Trace Properties: Properties of **2** traces

- Non-Interference
- Dependency

N-Trace Properties: Meta-dependencies

Program Properties - Summary

II. Self-Composition + Symbolic Execution

SELF-COMPOSITION – THE MAIN IDEA

G. Barthes

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

P. D'Argenio

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

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

C - a transpiler that computes the self-composition of s

T. Rezk

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 pr… scratch to check non-interference!

We are going to use symbolic execution

Self-Composition – example 1

```
l := h
```

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

We don't have to know! We can use symbolic execution…

Does the **assertion** hold?

Self-Composition – Example 1

```
l := h
```

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

We are going to execute the generated program symbolically

Instead of using concrete values, we use **symbolic variables**

SELF-COMPOSITION – EXAMPLE 1

```
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
```

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

```
True, [ l1→#l1, h1→#h1, l2→#l2, h2→#h2]
        ↓ assume(l1 = l2)
#l1=#l2, [ l1→#l1, h1→#h1, l2→#l2, h2→#h2]
        ↓ l1 := h1
#l1=#l2, [ l1→#h1, h1→#h1, l2→#l2, h2→#h2]
        ↓ l2 := h2
#l1=#l2, [ l1→#h1, h1→#h1, l2→#h2, h2→#h2]
```

SELF-COMPOSITION – EXAMPLE 1

```
l := h
```

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

`l2 := h2`

`#l1=#l2, [ l1→#h1, h1→#h1, l2→#h2, h2→#h2]`

`assert(l1 = l2)`

✗

$(\#l1 = \#l2) \Rightarrow (\#h1 = \#h2)$ **Valid?**

$(\#l1 = \#l2) \land (\#h1 \neq \#h2)$ **SAT?**

Self-Composition – Example 1

```
l := h
```

⇓

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

`l2 := h2`

`#l1=#l2, [ l1→#h1, h1→#h1, l2→#h2, h2→#h2]`

`assert(l1 = l2)`

✘

$(\#l1 = \#l2) \wedge (\#h1 \neq \#h2)$ **SAT?**

YES! [#l1→0, #h1→0, #l2→0, #h2→1]

Self-Composition – Example 2

```
if(h){
  l := 1
} else {
  skip
}
```

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

Self-Composition – Example 2

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

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

↓ assume(l1 = l2)

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

if (h1)

#h1 = 0 → ⋮

#h1 ≠ 0 → **Next Slide**

Self-Composition – Example 2

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

```
#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]
```

Self-Composition – Example 2

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

l1 := 1

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

if (h2)

#h2 = 0

…

#h2 ≠ 0

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

Self-Composition – Example 2

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

```
#h2 = 0              if (h2)
  ...  <-------------|
                     |
                     |  #h2 ≠ 0
                     v
#l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 ≠ 0,
[ l1→1, h1→#h1, l2→#l2, h2→#h2]
                     |
                     |  l2 := 1
                     v
#l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 ≠ 0,
[ l1→1, h1→#h1, l2→1, h2→#h2]
```

Self-Composition – Example 2

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

`l2 := 1`

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

Let's try another path

`assert(l1 = l2)`

`#l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 ≠ 0 ∧ 1 ≠ 1` **SAT?**

NO! No bug found

Self-Composition – Example 2

```
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

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

$\downarrow$ skip

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

$\downarrow$ assert(l1 = l2)

#l1=#l2 ∧ #h1 ≠ 0 ∧ #h2 = 0 ∧ 1 ≠ #l2 **SAT?**

YES! [#l1→0, #h1→1, #l2→0, #h2→0]

SELF-COMPOSITION – EXAMPLE 2

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

#h1 = 0

#h1 ≠ 0

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

#h1 ≠ 0,
[l1→#l1, h1→#h1,
l2→#l2, h2→#h2]

#h2 = 0

#h2 ≠ 0

#h2 = 0

#h2 ≠ 0

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

✓

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

✗

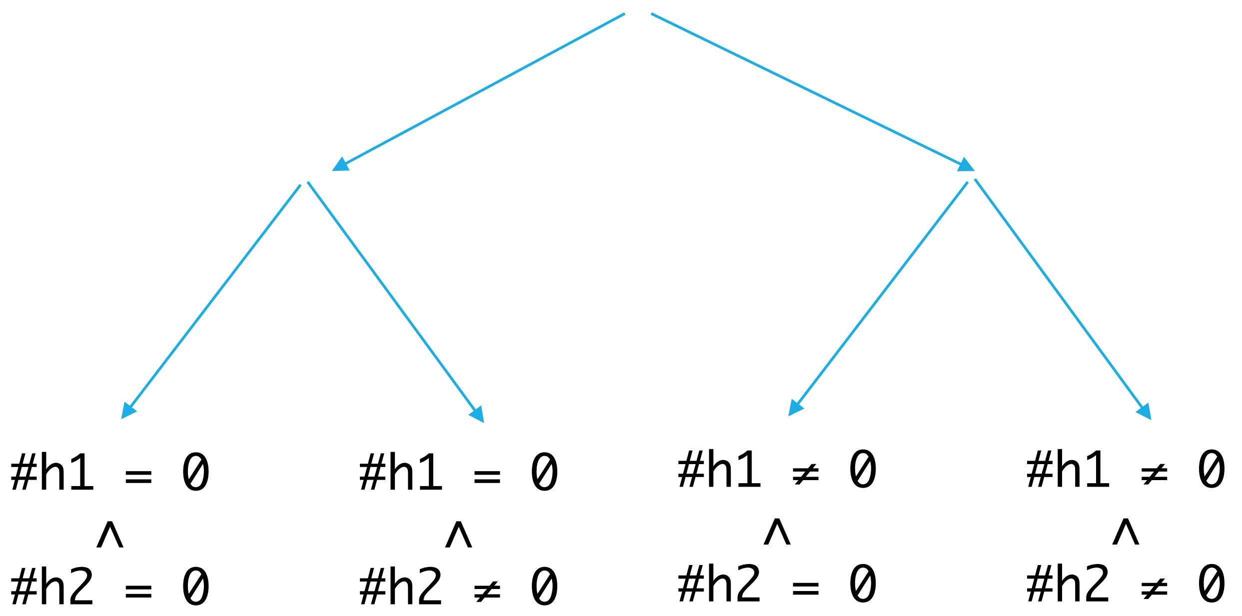
#h1 = 0 ∧ #h2 = 0,
[l1→**1**, h1→#h1,
l2→#l2, h2→#h2]

✗

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

✓

SELF-COMPOSITION – EXAMPLE 2

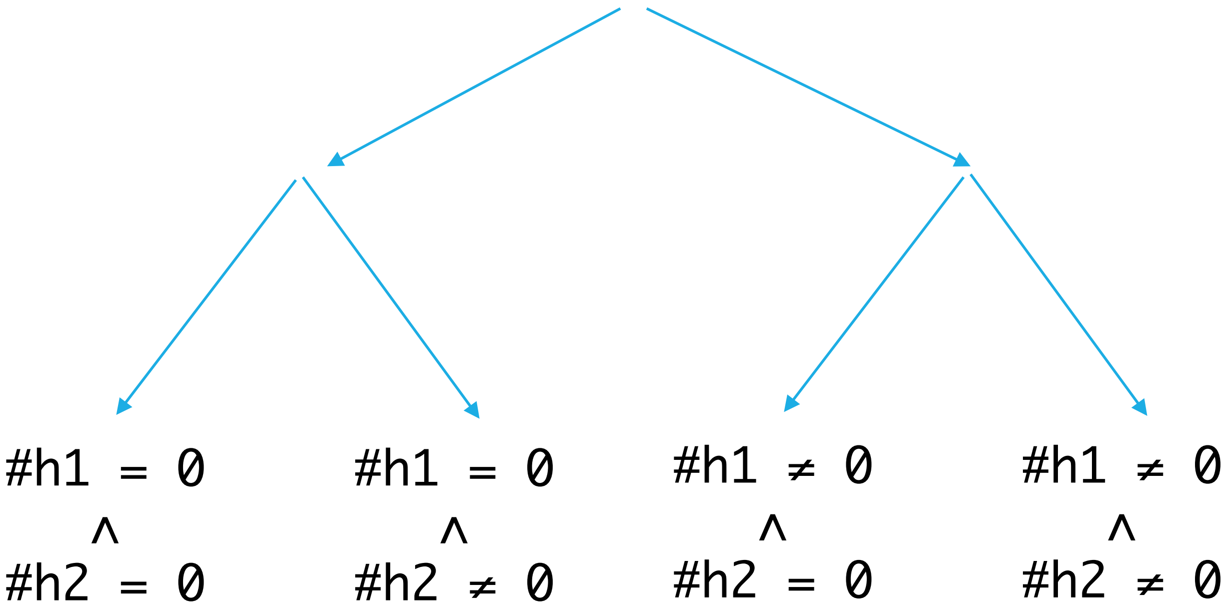


If we find a bug, we know that the program is **not** secure

But what if we do **not** find a bug?

Is the program secure?

Self-Composition – Example 2

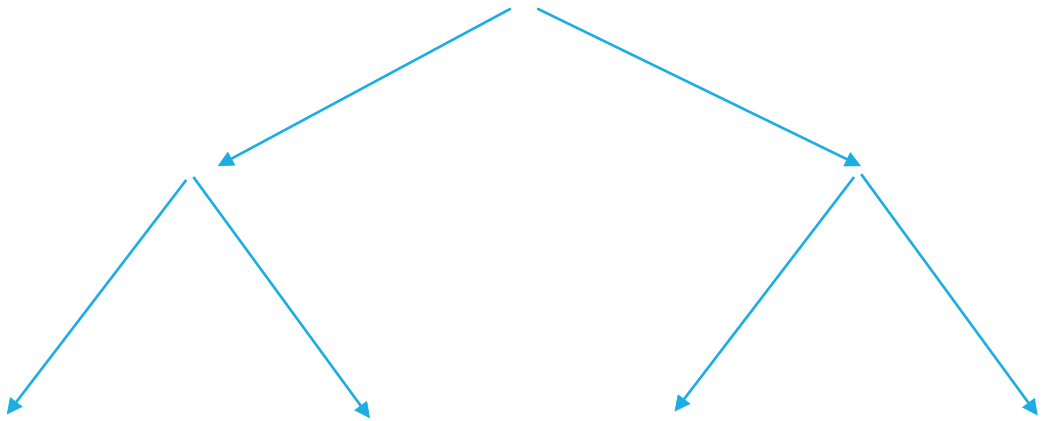


But what if we do **not** find a bug?

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

How can we know that?

SELF-COMPOSITION – EXAMPLE 2



$$(\#h1 = 0 \wedge \#h2 = 0) \vee (\#h1 = 0 \wedge \#h2 \neq 0) \vee (\#h1 \neq 0 \wedge \#h2 = 0) \vee (\#h1 \neq 0 \wedge \#h2 \neq 0) = \mathbf{True}$$

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

The disjunction of all final path conditions must be True

Self-Composition – Example 3

```
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)
```

SELF-COMPOSITION – EXAMPLE 3

```
#h1 = 0        #h1 = 0        #h1 ≠ 0        #h1 ≠ 0
   ∧              ∧              ∧              ∧
#h2 = 0        #h2 ≠ 0        #h2 = 0        #h2 ≠ 0
```

[l1→**1+#y1**, l2→**1+#y2**, h1→#h1, h2→#h2,
y1→#y1, y2→#y2, z1→#z1, z2→#z2]

The final symbolic store is always the same!

What is the SAT query?

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 Γ

$\mathcal{C}$ - a transpiler that computes the self-composition of s

Self-Composition – Formally

$$\frac{\mathrm{dom}(\theta_1) = \mathrm{dom}(\theta_2) = \mathrm{vars}(s) \qquad \mathrm{rng}(\theta_1) \cap \mathrm{rng}(\theta_2) = \emptyset}{C(s) \triangleq \theta_1(s); \theta_2(s)}$$

$$\begin{aligned}\mathcal{T}(\Gamma) \triangleq \{[\langle \rho_0, s_0 \rangle, \ldots, \langle \rho_n, s_n \rangle] \\ &\mid s_n = \mathsf{skip} \ \wedge \ \forall_{0 \le i < n} \langle \rho_i, s_i \rangle \to \langle \rho_{i+1}, s_{i+1} \rangle \\ &\wedge \Big(\bigwedge \{ (\rho_0(\theta_1(x)) = \rho_0(\theta_2(x))) \mid \Gamma(x) = L \} \\ &\quad \Rightarrow \bigwedge \{ (\rho_n(\theta_1(x)) = \rho_n(\theta_2(x))) \mid \Gamma(x) = L \} \Big) \}\end{aligned}$$

Self-Composition – Formally

$$\frac{\mathrm{dom}(\theta_1) = \mathrm{dom}(\theta_2) = \mathrm{vars}(s) \qquad \mathrm{rng}(\theta_1) \cap \mathrm{rng}(\theta_2) = \emptyset \\ s_{assume} = \mathrm{assume}\left(\bigwedge \{(\theta_1(x) = \theta_2(x)) \mid \Gamma(x) = L\}\right) \\ s_{assert} = \mathrm{assert}\left(\bigwedge \{(\theta_1(x) = \theta_2(x)) \mid \Gamma(x) = L\}\right)}{C(s) \triangleq s_{assume}; \theta_1(s); \theta_2(s); s_{assert}}$$

IV. Concolic Symbolic Execution

Concolic Symbolic Execution – the main idea

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

P. Godefroid

Why?

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

Concolic Symbolic Execution – the main idea

Concolic Testing:

Main Algorithm

$Input_0$ = *Pick random vector*

Coverage = **False**

i = 0

While ($Input_i \neq$ **NULL**) {

  (RES_i, PC_i)= Run Program with $Input_i$

  Coverage = Coverage $\vee PC_i$

  $Input_{i+1} \in$ **Models**($\neg$ Coverage)

  i = i+1

}

Concolic Symbolic Execution – the main idea

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

Step 1:

Inputs$_0$ = [x→22,y→7]
(RES$_1$, PC$_1$) = (OK, (x ≠ 2*y))
Coverage = (x ≠ 2*y)
Inputs$_1$ = [x→2,y→1]

Concolic Symbolic Execution – the main idea

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

Step 2:

```
Inputs₁ = [x→2,y→1 ]
(RES₂, PC₂) = (OK,(x = 2*y)∧(x<= y+10))
Coverage = (x ≠ 2*y) ∨
               ((x = 2*y)∧(x<= y+10))
Inputs₂ = [x→30,y→15 ]
```

CONCOLIC SYMBOLIC EXECUTION – THE MAIN IDEA

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

Step 3:

Inputs$_2$ = [x→30,y→15]
(RES$_3$, PC$_3$) = (**FAIL**,(x = 2*y)∧(x>y+10))
Coverage = (x ≠ 2*y) ∨
((x = 2*y)∧(x<=y+10)) ∨
((x = 2*y)∧(x>y+10))
Inputs$_2$ = **NULL**

A LOT MORE TO COVER...

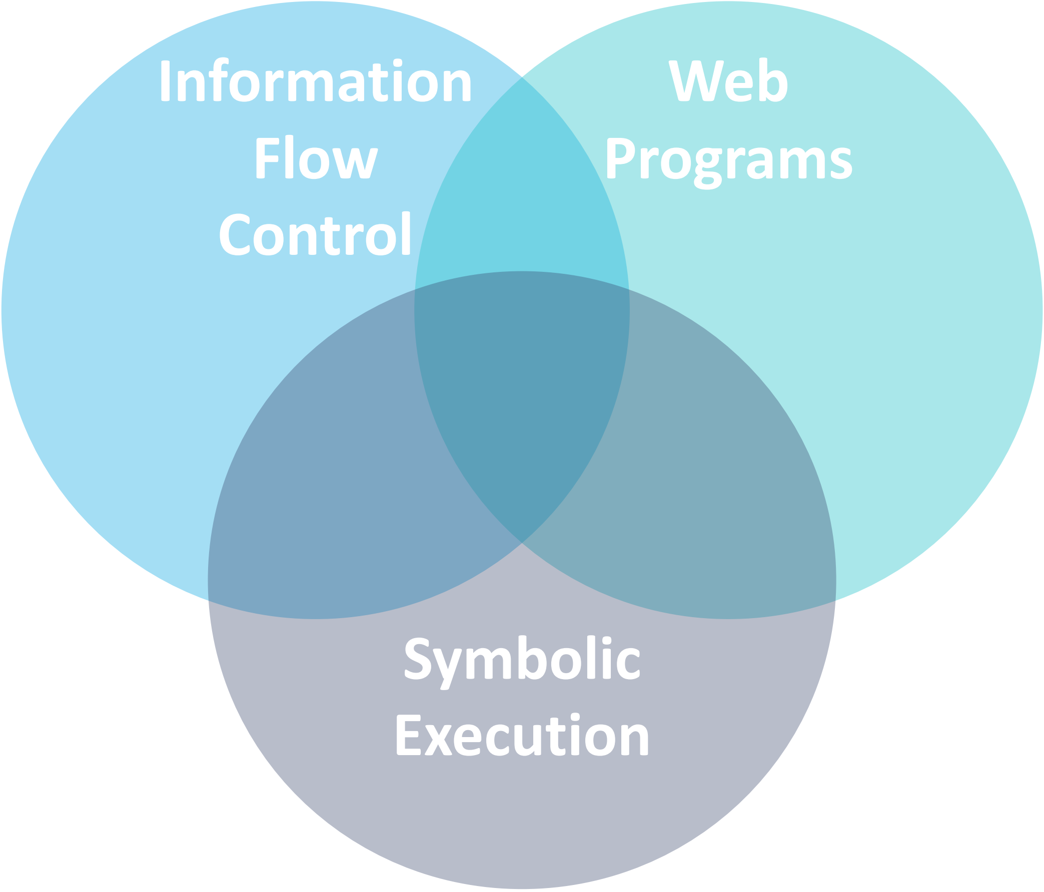
1. Symbolic execution with **data structures**
    - **Lazy-Initialization**
    - Data-structure **unfolding**

2. **Declassification**

3. Other Security Properties: **Confinement**

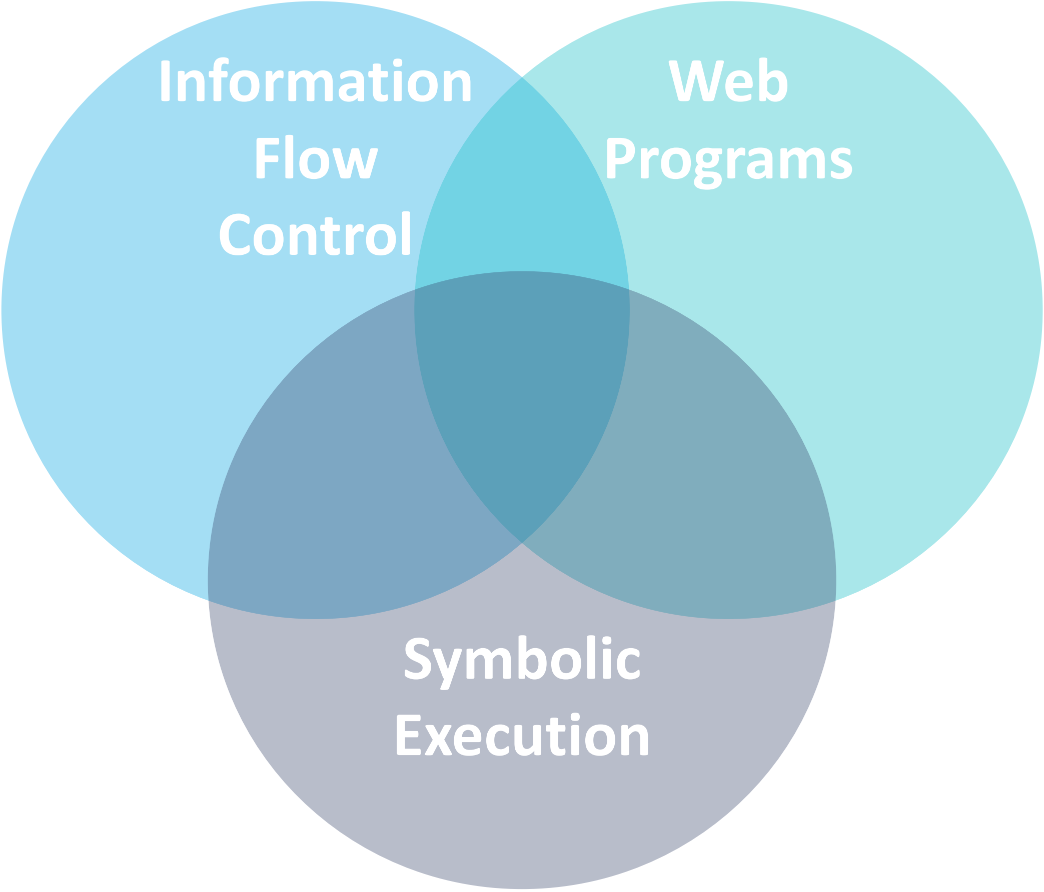
4. **Invariants and verification**

Master Projects



- 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

MASTER PROJECTS

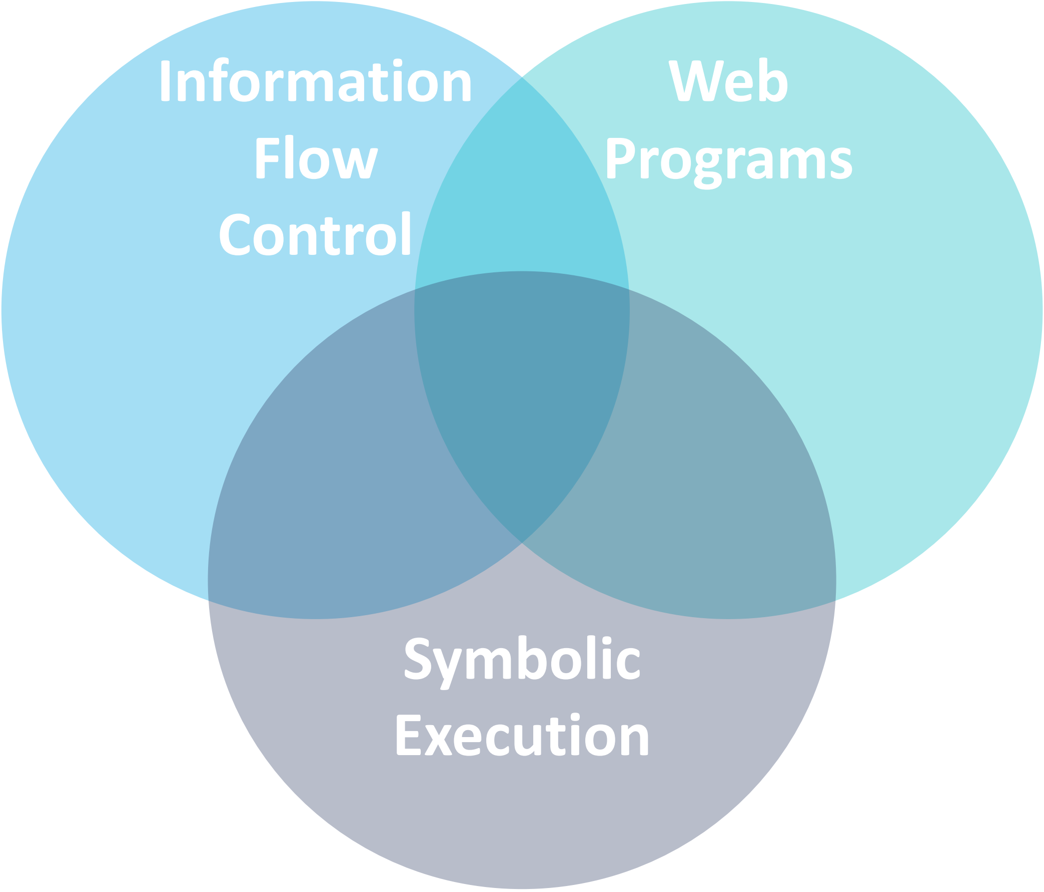


- Bounded model checking for JavaScript regular expressions
- Symbolically debugging secure information flow in the browser

And **more…** Check my website:

http://web.ist.utl.pt/**jose.fragoso**

Master Projects



Potential collaboratons with:

Imperial College London

Inria

IBM WATSON