Secure Multi-Execution

Dominique Devriese Frank Piessens

K.U.Leuven

May 14, 2010

Outline

Introduction

- Information Flow Analysis has received much attention:
  - Static analysis methods:
    From Denning to JFlow/JIF and FlowCaml
    But:
    - Substantial Programmer Effort
    - In general undecidable statically
    - Hard to handle exceptions, parallellism, timing covert channel
  - Dynamic methods:
    Many practical but unsound methods, some sound and somewhat practical methods
    But:
    - Some use cases require sound methods (e.g. web page scripts)
    - No existing monitor can precisely enforce non-interference
    - Hard to handle exceptions, parallellism, timing covert channel

Secure Multi-Execution

Secure Multi-Execution

- A novel dynamic enforcement technique for non-interference
- Nice theoretical properties
  - Strong soundness guarantee
  - The first (afawk) sound and precise enforcement method
- Practical in some scenario's
  - Performance measurements
  - Browser implementation possible?

Outline

Information Flow Analysis

```
1 var text = document.getElementById
2         ('email-input').text;
3 var abc = 0;
4 if(text.indexOf('abc')!=-1) { abc = 1 };
5 var url = 'http://example.com/img.jpg'
6     + '?t=' + escape(text) + abc;
7 document.getElementById('banner-img')
8     .src = url;
```

Information Flow Analysis

```
1 var text = document.getElementById
2         ('email-input').text; ⟵ Input at level H
3 var abc = 0;
4 if(text.indexOf('abc')!=-1) { abc = 1 };
5 var url = 'http://example.com/img.jpg'
6     + '?t=' + escape(text) + abc;
7 document.getElementById('banner-img')
8     .src = url; ⟵ Output at level L
```

Timing Covert Channel

```
1  function time(f) {
2    var t = new Date().getTime();
3    f();
4    return new Date().getTime() - t;
5  }
6  function f() {
7    if(abc != 0) {
8      for(var i = 0; i < 10000; ++i) {}
9    }
10 }
11 var abcLo = 0
12 if(time(f) > 10) {
13   abcLo = 1;
14 }
```

Termination Covert Channel

```
1 while(abc == 0) {}
2 img.url = 'http://example.com/img.jpg';
```

Secure Multi-Execution

L

```
1 var t = (...).text
2 var abc = 0
3 if(t.indexOf('abc')!=-1)
4   { abc = 1 }
5 var url = baseUrl + '?t='
6   + escape(t) + abc
7 (...).src = url
```

H

```
1 var t = (...).text
2 var abc = 0
3 if(t.indexOf('abc')!=-1)
4   { abc = 1 }
5 var url = baseUrl + '?t='
6   + escape(t) + abc
7 (...).src = url
```

Secure Multi-Execution

L

undefined

```
1 var t = (...).text
2 var abc = 0
3 if(t.indexOf('abc')!=-1)
4   { abc = 1 }
5 var url = baseUrl + '?t='
6   + escape(t) + abc
7 (...).src = url
```

H

```
1 var t = (...).text
2 var abc = 0
3 if(t.indexOf('abc')!=-1)
4   { abc = 1 }
5 var url = baseUrl + '?t='
6   + escape(t) + abc
7 (...).src = url
```

Secure Multi-Execution

L

undefined

```
1 var t = (...).text
2 var abc = 0
3 if(t.indexOf('abc')!=-1)
4   { abc = 1 }
5 var url = baseUrl + '?t='
6   + escape(t) + abc
7 (...).src = url
```

H

```
1 var t = (...).text
2 var abc = 0
3 if(t.indexOf('abc')!=-1)
4   { abc = 1 }
5 var url = baseUrl + '?t='
6   + escape(t) + abc
7 (...).src = url
```

Input Side Effects

L

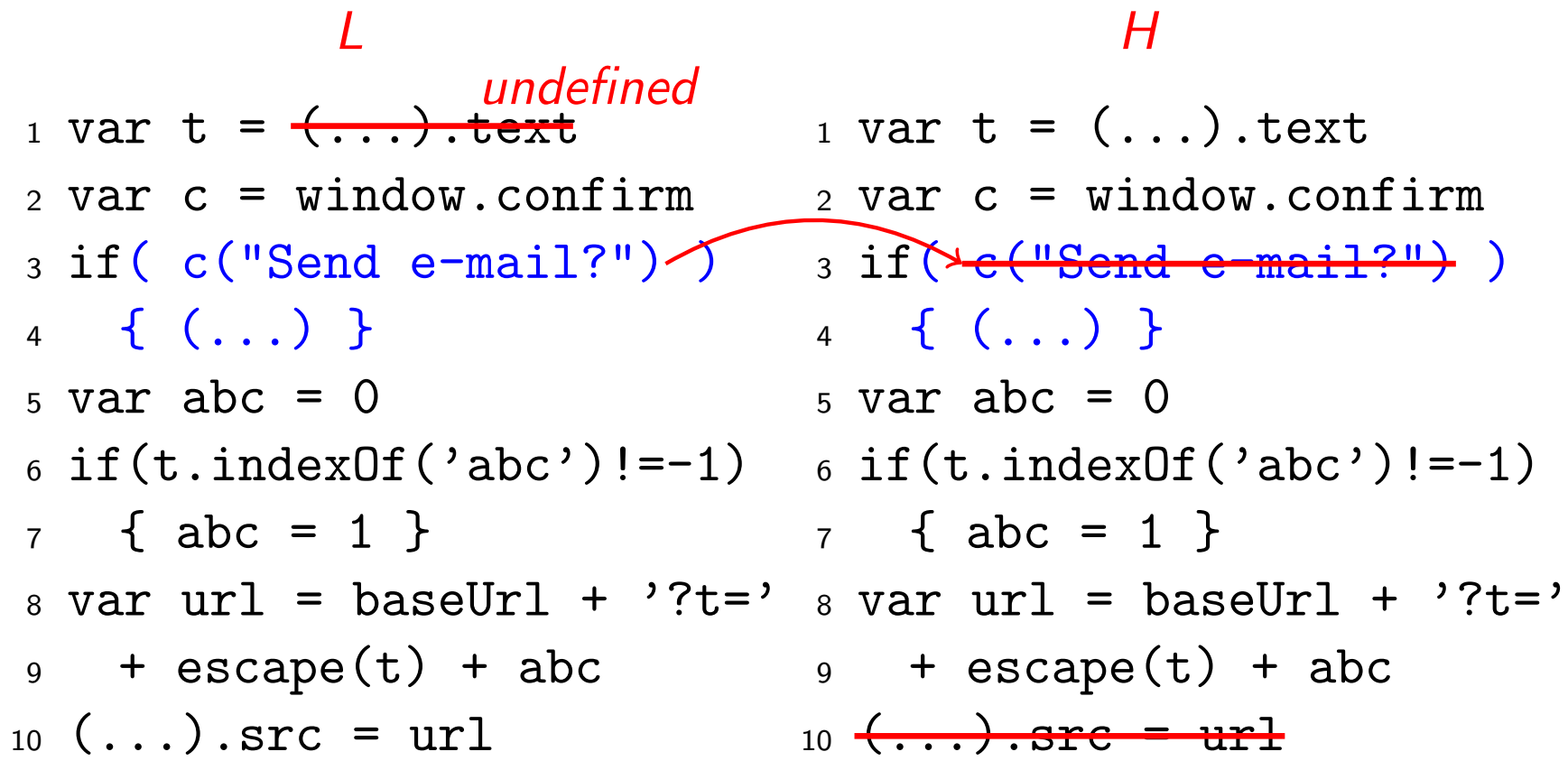
undefined

```
1  var t = (...).text
2  var c = window.confirm
3  if( c("Send e-mail?") )
4    { (...) }
5  var abc = 0
6  if(t.indexOf('abc')!=-1)
7    { abc = 1 }
8  var url = baseUrl + '?t='
9    + escape(t) + abc
10 (...).src = url
```

H

```
1  var t = (...).text
2  var c = window.confirm
3  if( c("Send e-mail?") )
4    { (...) }
5  var abc = 0
6  if(t.indexOf('abc')!=-1)
7    { abc = 1 }
8  var url = baseUrl + '?t='
9    + escape(t) + abc
10 (...).src = url
```

Input Side Effects



Secure Multi-Execution

Properties

- "Obviously" sound:
  Only execution at high level can see the real high inputs
  Only execution at low level can produce low outputs
- "Obviously" precise:
  If a program is non-interferent, then changing high inputs in low executions will not change their low behaviour

Outline

Non-interference

A formalization of information flow policies

Assume:

- sets of input channels $\mathcal{C}_i$, output channels $\mathcal{C}_o$
- security level lattice $\mathcal{L}$
- $\sigma_{in} : \mathcal{C}_i \to \mathcal{L}$, $\sigma_{out} : \mathcal{C}_o \to \mathcal{L}$
- inputs $I : \mathcal{C}_i \to (\mathbb{N} \to Int)$, outputs $O : \mathcal{C}_o \to List[Int]$
- $I =_l I'$ iff $I(c_i) = I'(c_i)$ for all c_i such that $\sigma_{in}(c_i) \leq l$

Definition

A program P is (termination-insensitively) *non-interferent* if for all security levels l and inputs $I =_l I'$, where P terminates for I and I' with outputs O and O', we have that $O =_l O'$.

Soundness

Definition (Strong non-interference)

A program P is *timing-sensitively non-interferent* or *strongly non-interferent* with relation to a given semantics $\hookrightarrow^*$ if for all security levels $l \in \mathcal{L}$, for all $n \geq 0$, for all program inputs I and I' such that $I =_l I'$ holds that if

$$(P, I) \hookrightarrow^n (p, O) \ ,$$

then

$$(P, I') \hookrightarrow^n (p', O') \ ,$$

and $p' =_l p$ and $O' =_l O$.

Theorem (Soundness of Secure Multi-Execution)

Any program P is strongly non-interferent under secure multi-execution, using the $\texttt{select}_{\text{lowprio}}$ *scheduler function.*

Precision

Theorem (Precision of Secure Multi-Execution)

Suppose we have a (termination-sensitively) non-interferent program P. Suppose that

$$(P, I) \rightarrow^* (p, O)$$

(terminates) for some I, p and O. Then

$$(P, I) \Rightarrow^* (p, O) \ .$$

Outline

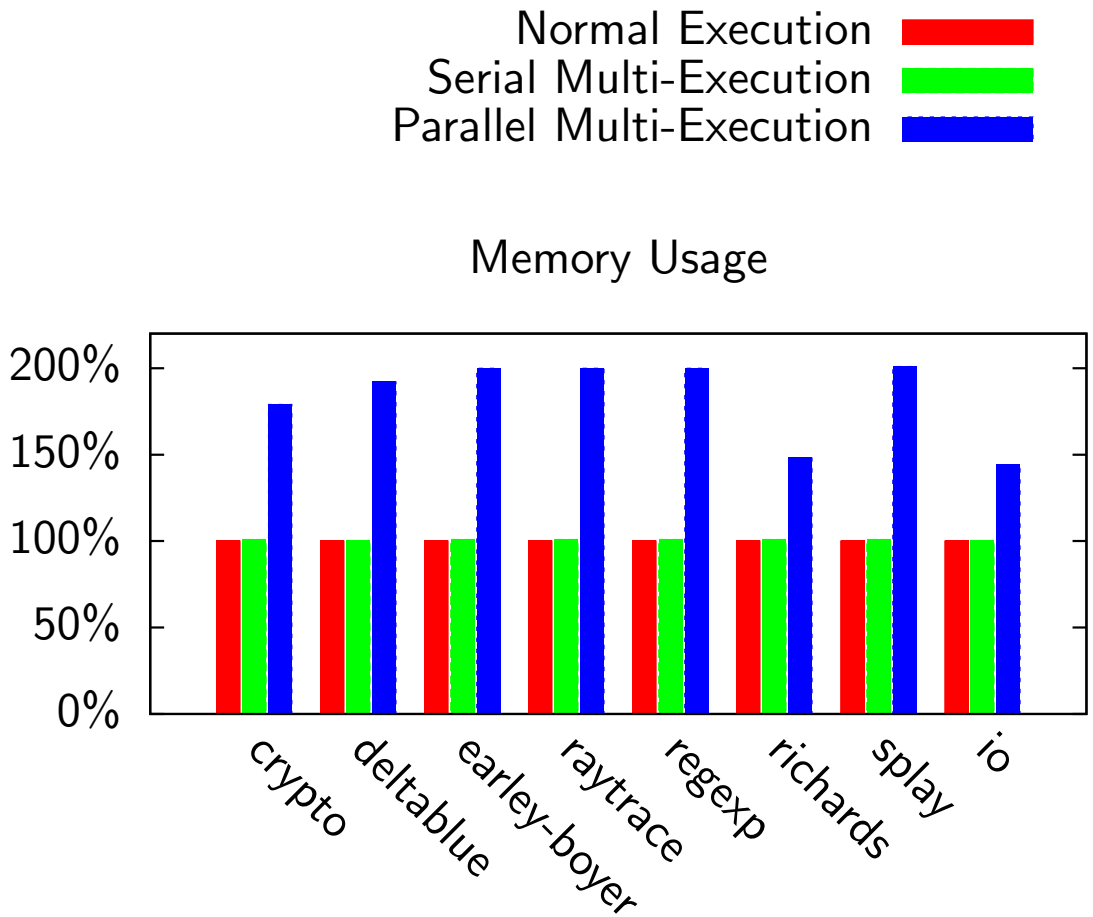
Experimental Results

- Spidermonkey Javascript engine, no real browser
- 2 security levels
- Dual-core PC
- 3 types of execution:
  - Standard execution
  - Serial Multi-Execution
  - Parallel Multi-Execution
- Benchmarks:
  - Google Chrome V8 Benchmark Suite: crypto, deltablue, earley-boyer, raytrace, regexp, richards, splay
  - io: model I/O functions: hi_input, hi_output, lo_input, lo_output: some calculations + I/O at different security levels

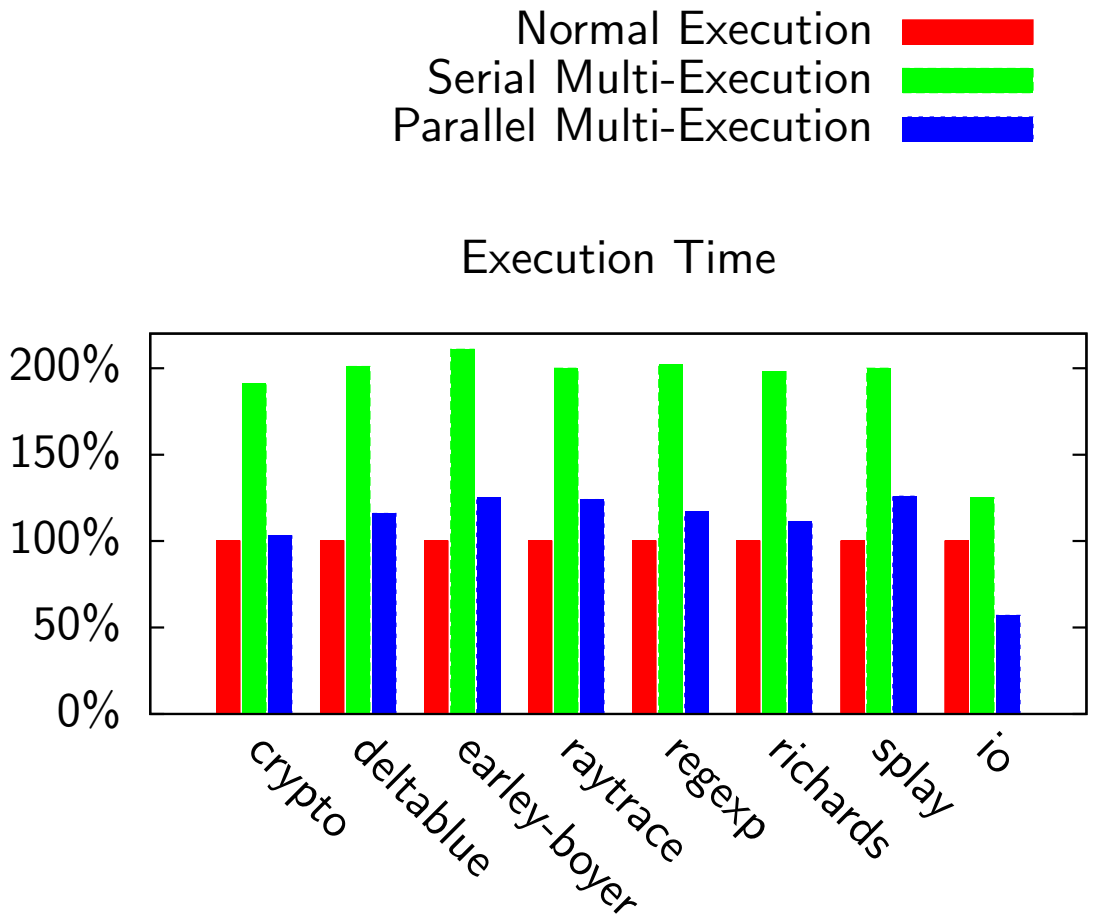
IO Benchmark

```
1  for (var i = 0; i < 100; ++i) {
2    var test = 0;
3    for (var j = 0; j < 10000; ++j) {
4      test += j;
5    }
6    if (i % 10 == 0) {
7      var hi_in = hi_input();
8      var lo_in = lo_input();
9      lo_output("#" + i + ". lo_in: '"
10       + lo_in + "'. hi_in is: '"
11       + hi_in + "'");
12     hi_output("#" + i + ". hi_in: '"
13       + hi_in + "'. lo_in is: '"
14       + lo_in +"'");
15   }
16 }
```

Experimental Results: Memory Usage

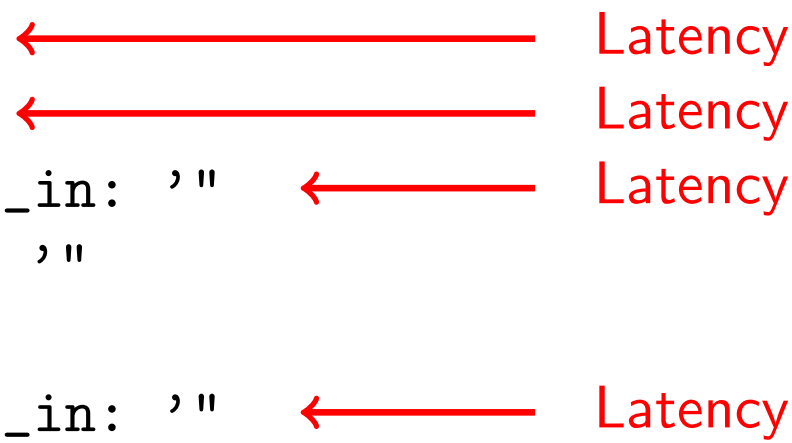


Experimental Results: Execution Time



Automatic parallellization

```
1  for (var i = 0; i < 100; ++i) {
2    var test = 0;
3    for (var j = 0; j < 10000; ++j) {
4      test += j;
5    }
6    if (i % 10 == 0) {
7      var hi_in = hi_input();
8      var lo_in = lo_input();
9      lo_output("#" + i + ". lo_in: '"
10       + lo_in + "'. hi_in is: '"
11       + hi_in + "'");
12     hi_output("#" + i + ". hi_in: '"
13       + hi_in + "'. lo_in is: '"
14       + lo_in +"'");
15   }
16 }
```



Automatic parallellization

```
1  for (var i = 0; i < 100; ++i) {
2    var test = 0;
3    for (var j = 0; j < 10000; ++j) {
4      test += j;
5    }
6    if (i % 10 == 0) {
7      var hi_in = hi_input();
8      var lo_in = lo_input();
9      lo_output("#" + i + ". lo_in: '"
10       + lo_in + "'. hi_in is: '"
11       + hi_in + "'");
12     hi_output("#" + i + ". hi_in: '"
13       + hi_in + "'. lo_in is: '"
14       + lo_in +"'");
15   }
16 }
```

	L	H
←	Skip	Latency
←	Latency	Reuse
←	Latency	Skip
←	Skip	Latency

Outline

The technique's merits

Advantages:

Very strong	Soundness guarantee
Very general	No fundamental issues with parallellism, exceptions or other language features
Good precision	No change for (termination-sensitively) non-interferent programs
Acceptable imprecision	Interferent executions are modified in acceptable way (intuitive, no formalisation...)
Dynamic	Run-time assignment of I/O channels to security levels

Downsides:

Performance	Acceptable for some use cases?

Thank you for your attention.

Any questions?