Noninterference Through Secure Multi-Execution

Dominique Devriese
DistriNet Research Group, KULeuven
E-mail: dominique.devriese@cs.kuleuven.be

Frank Piessens
DistriNet Research Group, KULeuven
E-mail: frank.piessens@cs.kuleuven.be

***Abstract*—A program is defined to be noninterferent if its outputs cannot be influenced by inputs at a higher security level than their own. Various researchers have demonstrated how this property (or closely related properties) can be achieved through information flow analysis, using either a static analysis (with a type system or otherwise), or using a dynamic monitoring system.**

We propose an alternative approach, based on a technique we call *secure multi-execution*. The main idea is to execute a program multiple times, once for each security level, using special rules for I/O operations. Outputs are only produced in the execution linked to their security level. Inputs are replaced by default inputs except in executions linked to their security level or higher. Input side effects are supported by making higher-security-level executions reuse inputs obtained in lower-security-level threads.

We show that this approach is interesting from both a theoretical and practical viewpoint. Theoretically, we prove for a simple deterministic language with I/O operations, that this approach guarantees complete soundness (even for the timing and termination covert channels), as well as good precision (identical I/O for terminating runs of termination-sensitively noninterferent programs).

On the practical side, we present an experiment implementing secure multi-execution in the mainstream Spidermonkey Javascript engine, exploiting parallelism on a current multi-core computer. Benchmark results of execution time and memory for the Google Chrome v8 Benchmark suite show that the approach is practical for a mainstream browser setting. Certain programs are even executed faster under secure multi-execution than under the standard execution.

We discuss challenges and propose possible solutions for implementing the technique in a real browser, in particular handling the DOM tree and browser callback functions. Finally, we discuss how secure multi-execution can be extended to handle language features like exceptions, concurrency or nondeterminism.

I. Introduction

The identification and suppression of illegal information flows in software programs has been an active research topic for several decades. Roughly speaking, existing approaches can be classified as *static* techniques based on type systems [1]–[3], abstract interpretation [4] or other static analysis methods [5], *dynamic* techniques based on execution monitoring [6], [7], or combinations of static and dynamic techniques [8].

However, while progress in both static and dynamic approaches has been impressive, there are fundamental limits on what can be achieved with a combination of static analysis and execution monitoring. It is easy to show that noninterference (the absence of information flow from public inputs to secret outputs) is undecidable statically, and it has been shown that execution monitoring can not enforce noninterference precisely [9].

In this paper, we propose a novel approach based on a technique we call *secure multi-execution*. The main idea is to execute a program multiple times, once for each security level, using special rules for I/O operations. Outputs on a given output channel are only produced in the execution linked to the security level assigned to this output channel. Inputs from a given input channel are replaced by default inputs except in executions linked to a security level higher than or equal to the level of this input channel. Input side effects are supported by making higher-security-level executions reuse inputs obtained in lower-security-level threads.

Secure multi-execution has strong advantages. First, it is *sound*: any program is noninterferent under secure multi-execution. This is relatively easy to see: an execution at a given security level can only output at that level, and never even see inputs from a higher level. So outputs could not possibly depend on inputs from higher levels. Second, secure multi-execution is *precise*: if a program is (termination-sensitively) noninterferent under normal execution, then its behaviour under a terminating normal execution and under secure multi-execution are the same. As discussed above, techniques based on static analysis and/or execution monitoring can not achieve both soundness and precision. We believe our enforcement mechanism is the first one that is both provably sound and *precise* in the sense that the enforcement mechanism is transparent for terminating runs of *every* termination-sensitively noninterferent program (a notion we will define later).

One obvious disadvantage of multi-execution is its cost in terms of CPU time and memory use. However, we argue in this paper for the practicality of the approach in at least one important application area:

Javascript web applications. We have implemented secure multi-execution in the mainstream Spidermonkey Javascript interpreter [10], and we show using the Google Chrome v8 benchmarks [11] that memory costs and CPU time costs can be traded off against each other. More surprisingly, we show that secure multi-execution can actually be *faster* than standard execution. This is due to the fact that Javascript is a sequential language and hence necessarily serialises all I/O operations. Parallel multi-execution provides more opportunity for I/O parallelism, and this can lead to a faster execution time. One could say that secure multi-execution uses the noninterference property of a program to automatically parallelise the program.

In summary, this paper makes the following contributions:

- We propose *secure multi-execution* as a new enforcement mechanism for noninterference. We propose rules for correctly handling a generic form of I/O.
- We formally prove soundness and precision of secure multi-execution for a small model language with I/O operations.
- We present experimental benchmark results for multi-execution implemented in a mainstream Javascript engine. This makes secure multi-execution to our knowledge the first provably sound dynamic noninterference-enforcing technique for which benchmark results are available.
- We argue for the practicality of multi-execution in a mainstream browser setting, based on our benchmark results and outlines of proposed solutions for handling the DOM tree and browser callbacks.

The remainder of this paper is structured as follows. First, in Section II we give a motivation and informal overview of our approach. Next, in Section III we provide a formal model, and we prove soundness and precision in Section IV. Section V then reports on our implementation and the experiments we performed. In Sections VI, VII and VIII we discuss further extensions, related work and offer a conclusion.

II. Informal overview

A. Information Flow Analysis

Imagine the following Javascript code is running in a typical web-based e-mail client.

```
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;
```

Note that setting the `src` property of an img element will trigger an HTTP request to the url set. Clearly, this code represents an important breach of privacy. The owners of the example.com domain can obtain the body of any e-mail a user sends. This type of code could have been injected through a cross-site scripting (XSS) attack, or hidden in an ad included by the web-mail server into the application. Many authors have discussed this vulnerability in current browsers, and various countermeasures have been studied in the research literature [7], [12]–[14].

A first step at analysing the above program is to classify inputs and outputs into security levels. For example, the expression "`document.getElementById('email-input').text`" could be seen as an input at security level H (*high, confidential*). The expression "`document.getElementById('banner-img').src`" could be seen as an output at security level L (*low, public*). The example above exhibits a flow of information from a H input into a L output. Note that in this classification, it is important to ensure that no input or output can affect subsequent inputs that are not on higher or equal security levels.

Countermeasures are often shown to be effective at eliminating these unacceptable flows by showing that they guarantee a property called *noninterference*, which comes in different variants. *Termination-insensitive noninterference* is a common variant, guaranteeing that two terminating executions of a program produce output that agrees on public data when started with input that agrees on public data. The assumption behind this notion is that information is only disclosed by the program when it terminates.

Termination-insensitive noninterference forbids two types of information flow, both of which are present in the example above. The variable `text` which is sent to the L output, was assigned data directly coming from the H input. This type of information flow is commonly referred to as an *explicit flow*. The variable `abc` was not directly assigned data from the H input, but it was assigned a value in a conditional branch on a condition involving H variables. This type of information flow is called an *implicit flow*. A program cannot be termination-insensitively noninterferent if these types of flows are present.

This variant of noninterference can thus already offer some protection, but there are still ways to bypass it using what are generally called *covert channels*. Figure 1 shows two programs exploiting respectively the timing and termination covert channels. Using either program, an attacker can get hold of the value of the variable `abc` above.

```
function time(f) {
  var t = new Date().getTime();
  f();
  return new Date().getTime() - t;
}
function f() {
  if(abc != 0) {
    for(var i = 0; i < 10000; ++i) {}
  }
}
var abcLo = 0
if(time(f) > 10) {
  abcLo = 1;
}
```

(a) Exploiting the timing covert channel.

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

(b) Exploiting the termination covert channel.

Figure 1. Bypassing information flow analysis using covert channels.

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

(a) Execution at L security level.

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

(b) Execution at H security level.

Figure 2. Multi-Execution of example Javascript program from Section II-A.

Using secure multi-execution, we can close the timing and termination covert channels as well. We guarantee a stronger notion, which we call *timing- and termination-sensitive noninterference*. This stronger notion states that *after any number of execution steps*, two executions of a program will have produced output which agrees on public data when run with input that agrees on public data.

B. Secure Multi-Execution

As its name suggests, secure multi-execution will execute a program multiple times, once for each security level. Statements producing externally observable output are only executed in the execution linked to their security level. Input at high security levels is replaced by the Javascript *undefined* value in executions at lower security levels. In Figure 2a and 2b, we illustrate how the example Javascript program from Section II-A would be executed on respectively the L and H security level. Recall that we had associated the expressions "`document.getElementById('email-input').text`" and "`document.getElementById('banner-img').src`" respectively to the H and L security levels. Modifications to the behaviour of input and output statements are indicated by crossing them out and, if necessary, putting a replacement expression on the side.

It is clear how multi-execution ensures noninterference. Output statements are only executed from an execution at the associated security level, and this execution simply does not get access to the actual values of inputs at higher security levels. Even the timing and termination covert channels are thus easily blocked. Because the program is executed once for every security level, all input and output statements are still executed and the effects of noninterferent programs should not be modified. An interesting question is how multi-execution will handle interferent programs. A partial answer is that it will always execute them in a noninterferent way, modifying their behaviour if necessary to ensure this. We discuss this further in Section VI-B.

Contrary to what the illustration in Figure 2 suggests, both in our formal work in Sections III and IV and in our experiment described in Section V, we do not implement secure multi-execution through program transformation. Instead, we run multiple executions in parallel, using a scheduler to determine the interleaving. We intercept input and output commands and execute them according to the rules we described, depending on the security level of the execution they are called from.

C. Input Side Effects

One problem that still remains is how multi-execution will handle inputs with side effects. For example, the following program uses the commonly supported browser `window.confirm` method:

```
1 var r = window.confirm("Are you sure?");
```

If we execute this program at multiple security levels, each higher than the one associated to the `window.confirm` input, then the user will need to reply multiple times to the question.

To solve this problem, we assume that each such input will correspond to exactly one identical input in the execution at the input's own security level. We will show in the formal derivations in Section IV that for terminating runs of termination-sensitively noninterferent programs, this assumption holds. We can therefore let the executions at higher levels wait for the execution at the lower level and reuse the input value collected there. In fact, this solution also solves the problem of exposing timing and termination channels related to the side effects of the input function.

Note, finally, that our inputs with side effects are in fact a more general concept than outputs. The outputs we will consider are limited to producing a certain side effect in the outside world and cannot return a result to the program. A statement producing both an effect in the outside world and returning a result must in fact be treated as an input statement, and its effects as side effects of producing the result.

D. Scheduling and Expected Performance

The basic idea of secure multi-execution is executing a program one time per security level. Clearly, this is not a cheap approach in terms of performance. First intuition would suggest that both execution time and memory usage would be multiplied by the number of security levels. In Section V, we provide detailed measurements of both execution time and memory usage in an experimental implementation of secure multi-execution in a Javascript engine. In this section, we already discuss some results informally.

The impact of secure multi-execution on execution time and memory usage depends heavily on the choice of a scheduling strategy for the different executions. The choice of a scheduling strategy is free, except for two important remarks. First, as we discussed above, high security executions need to wait for low security ones when they read from low security inputs. Second, if we want to completely close timing covert channels, it must not be possible for any low security execution to ever have to wait for a high security one. This does not mean that serialising the executions is the only option. Higher security execution can still be allowed to progress while lower security threads are waiting for I/O. On multi-core systems, more than one execution can progress at the same time.

A simple scheduling strategy is to run the different executions serially, running lower security executions first. This is an important strategy, corresponding to what we will later describe as the $\texttt{select}_{\text{lowprio}}$ scheduling function. In general, this strategy will multiply execution time by the number of security levels, but will not add significant memory usage overhead, because each previous execution is completely finished before the next one starts, so its memory can immediately be reclaimed.

$$
\begin{aligned}
\mathit{command} ::= {} & x := e \\
\mid {} & c; c \\
\mid {} & \textbf{if } e \textbf{ then } c \textbf{ else } c \\
\mid {} & \textbf{while } e \textbf{ do } c \\
\mid {} & \textbf{skip} \\
\mid {} & \textbf{input } x \textbf{ from } i \\
\mid {} & \textbf{output } e \textbf{ to } o
\end{aligned}
$$

Figure 3. Command syntax of our model language.

Probably a more practical strategy consists of running the different executions each in a parallel thread, giving priority to lower security ones. Especially on multi-core systems (which seem to be becoming standard on consumer PC's), this strategy will significantly reduce execution time overhead, at the expense of increased memory usage. We think that for typical web applications, this is a preferable compromise.

In Section V, we discuss a somewhat unexpected feature of secure multi-execution using this last scheduling strategy. For programs incurring I/O latency from I/O channels on different security levels, secure multi-execution can actually speed up a program's execution. In this situation, secure multi-execution functions as an automatic program paralleliser, based on the assumption of noninterference of a program.

III. Formalisation

A. Model language

We introduce a simple imperative model language and related concepts, to explain and prove the technical properties of our approach. Our model language and semantics are deterministic.

Our model language is fairly standard, and is based on the one used by Russo et al. [15]. We have removed threading primitives and added simple input and output commands. Values can be booleans or integers. We assume atomic, deterministic and side-effect-free expressions. A program P is a command intended for execution by the system. We define the language's command syntax in Figure 3.

We assume a set of input channels C_{in} and output channels C_{out}. We define a *program input* I as a mapping from input channels $i \in C_{\text{in}}$ to *channel input queues* q,

$$\frac{I(i) = q \qquad p(i) = n \qquad q(n) = v}{\text{read}\,(I, i, p) = v} \tag{1}$$

$$\frac{O(o) = [v_1, \cdots, v_n]}{\text{write}\,(O, o, v) = O\,[o \mapsto [v_1, \cdots, v_n, v]]} \tag{2}$$

Figure 4. Primitive read and write operation for working with program inputs and program outputs.

mapping non-negative integers to values. We define an *input pointer* p as a mapping from input channels $i \in C_{\text{in}}$ to integers. The symbol p_0 denotes an initial input pointer mapping every input channel to position 0 ($\_ \mapsto 0$). A *program output* O is defined as a mapping from output channels $o \in C_{\text{out}}$ to lists of values. The symbol O_0 denotes an initial program output mapping every output channel to the empty list ($\_ \mapsto []$). In Figure 4, we define primitive read and write operations on program inputs and program outputs.

We define standard small-step execution semantics of the language in Figure 3. These are defined for *execution configurations* $\langle c, m, p, I, O\rangle$, with c a command, m a memory (a function mapping variables to their values), p an input channel pointer, I a program input and O a program output.

We define $m\,[x \mapsto v]$ to represent a new memory mapping variable x to value v, and all other variables y to $m(y)$. We also extend memories m to expressions, so that $m(e) = v$ means that v is the value of the expression e evaluated with respect to the variable values in m. The notation m_0 denotes an initial memory mapping every variable to the value 0 ($\_ \mapsto 0$).

The complete semantics can be found in Figure 5. Note that standard semantics are represented using the symbol $\rightarrowtriangle$, in order to be able to easily distinguish them from the multi-execution semantics we will define later.

A program P can be executed with respect to a given initial program input I. It is executed by applying the semantic rules in Figure 5 to the execution configuration $\langle P, m_0, p_0, I, O_0\rangle$.

We write $\rightarrowtriangle^*$ for the transitive and reflexive closure of the $\rightarrowtriangle$ relation for execution configurations. A program P terminates for a given initial input I if $\langle P, m_0, p_0, I, O_0\rangle \rightarrowtriangle^* \langle \textbf{skip}, m_f, p_f, I, O_f\rangle$ for some final memory m_f, final input pointer p_f and final program output O_f. In such a case, we say that the execution of P for program input I produces final input pointer p_f and program output O_f. We write $(P, I) \rightarrowtriangle^* (p_f, O_f)$.

We also introduce a time-limited execution relation. We write $\langle c, m, p, I, O\rangle \rightarrowtriangle^n \langle c', m', p', I, O'\rangle$ iff execution configuration $\langle c, m, p, I, O\rangle$ can be transformed

$$\frac{c = \textbf{if } e \textbf{ then } c_{\text{true}} \textbf{ else } c_{\text{false}} \qquad m(e) = b}{\langle c, m, p, I, O\rangle \rightarrowtriangle \langle c_b, m, p, I, O\rangle} \tag{1}$$

$$\frac{\langle c_1, m, p, I, O\rangle \rightarrowtriangle \langle c_1', m', p', I, O'\rangle}{\langle c_1; c_2, m, p, I, O\rangle \rightarrowtriangle \langle c_1'; c_2, m', p', I, O'\rangle} \tag{2}$$

$$\frac{}{\langle \textbf{skip}; c, m, p, I, O\rangle \rightarrowtriangle \langle c, m, p, I, O\rangle} \tag{3}$$

$$\frac{c = \textbf{while } e \textbf{ do } c_{\text{loop}} \qquad m(e) = \text{true}}{\langle c, m, p, I, O\rangle \rightarrowtriangle \langle c_{\text{loop}}; c, m, p, I, O\rangle} \tag{4}$$

$$\frac{c = \textbf{while } e \textbf{ do } c_{\text{loop}} \qquad m(e) = \text{false}}{\langle c, m, p, I, O\rangle \rightarrowtriangle \langle \textbf{skip}, m, p, I, O\rangle} \tag{5}$$

$$\frac{m(e) = v \qquad m' = m\,[x \mapsto v]}{\langle x := e, m, p, I, O\rangle \rightarrowtriangle \langle \textbf{skip}, m', p, I, O\rangle} \tag{6}$$

$$\frac{\begin{array}{c} c = \textbf{output } e \textbf{ to } o \\ m(e) = v \qquad O' = \text{write}\,(O, o, v)\end{array}}{\langle c, m, p, I, O\rangle \rightarrowtriangle \langle \textbf{skip}, m, p, I, O'\rangle} \tag{7}$$

$$\frac{\begin{array}{c} c = \textbf{input } x \textbf{ from } i \qquad \text{read}\,(I, i, p) = v \\ p' = p[i \mapsto p(i) + 1] \qquad m' = m[x \mapsto v]\end{array}}{\langle c, m, p, I, O\rangle \rightarrowtriangle \langle \textbf{skip}, m', p', I, O\rangle} \tag{8}$$

Figure 5. Standard small-step semantics of the model language.

to execution configuration $\langle c', m', p', I, O'\rangle$ by n execution steps. If $\langle P, m_0, p_0, I, O_0\rangle \rightarrowtriangle^n \langle c', m', p', I, O'\rangle$, we write that $(P, I) \rightarrowtriangle^n (p', O')$.

Note also that the final memory m_f is assumed not to be publicly observable. As discussed by Le Guernic et al. [6] (who take the same view), public final variables x can if necessary be encoded by introducing an output channel o at an appropriate security level and by adding a command **output** x **to** o at the end of the program.

B. Secure Multi-Execution

In this section, we proceed to modeling secure multi-execution semantics. The main difficulty lies in the handling of inputs with side effects (see section II-C), requiring scheduling and synchronisation between the executions at different security levels. In order to model this, we will define secure multi-execution semantics on two levels. The *local* semantics model steps in a single execution at a certain security level, while the *global* semantics model the scheduling and synchronisation semantics needed between the separate local executions.

We assume a security level lattice $\mathcal{L}$ and functions $\sigma_{\text{in}} : C_{\text{in}} \to \mathcal{L}$ and $\sigma_{\text{out}} : C_{\text{out}} \to \mathcal{L}$, mapping each input channel i or output channel o to a security level. We

assume $\mathcal{L}$ to be finite. This can be achieved by limiting $\mathcal{L}$ to the security levels for which I/O statements are present in the program at hand. The lattice order on $\mathcal{L}$ can always be extended to a total order (since $\mathcal{L}$ is finite), and we assume as given one such extension.

Note that initial security levels for variables are not directly supported. However, we can encode a variable x with an initial security level l and initial value v by introducing a dummy input channel i_x, such that $\sigma_{\text{in}}(i_x) = l$ and using an input channel state I such that $I(i_x) = (\_ \mapsto v)$.

Let's proceed to modelling the separate executions at the different security levels. These different executions normally do not interact, except when the execution for an execution level l_1 reads from a channel i with $\sigma_{\text{in}}(i) = l_2 \leq l_1$. In that case, as explained in Section II-C, the execution at level l_1 will wait for the execution at level l_2 to read the next value from channel i and then use that value as well.

To model this, we define one *local execution configuration* for each security level, as well as a single *global execution configuration* describing the global state of the execution. A local execution configuration $\langle c, m, p\rangle_l$ is defined by a command c, a memory m, an input pointer p and a security level l. A global execution configuration $\langle [lec_1, \cdots, lec_n], wq, r, I, O\rangle$ is defined by a set of local execution configurations $[lec_1, \cdots, lec_n]$, a *waiting queue* wq, mapping pairs (i, n) to sets of local execution configurations, a global input pointer r, a program input I and a program output O.

In Figure 6, we define local semantics, modelling steps in a single execution at a security level l. The local semantics are defined for local execution states with respect to a global input pointer r, program input I and program output O. Local execution steps can emit a signal $\odot(i, n)$, indicating that the execution at security level $\sigma_{\text{in}}(i)$ has just read from channel i at position n. They can also emit a signal $\otimes(i, n)$ indicating that the execution cannot proceed until the execution at security level $\sigma_{\text{in}}(i)$ has read from channel i at position n.

Figure 7 shows the global semantics, modelling a scheduler and keeping track of global state. They are defined assuming a procedure `select`, which maps a list of local execution states $([lec_1, \cdots, lec_n])$ onto the execution state lec_i next to be run. We define `select` as a procedure, not a function, to allow for modelling nondeterministic schedulers and schedulers keeping state, like a round-robin scheduler.

The scheduler $\texttt{select}_{\text{lowprio}}$, which we need for proving one of our main results, is a function mapping a list of local execution states $([lec_1, \cdots, lec_n])$ onto the local execution state with minimal security level, according to the assumed total extension of the lattice order

$$\frac{c = \textbf{if } e \textbf{ then } c_{\text{true}} \textbf{ else } c_{\text{false}} \qquad m(e) = b}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle c_b, m, p\rangle_l, r, I, O} \tag{1}$$

$$\frac{\langle c_1, m, p\rangle_l, r, I, O \Rightarrow \left\langle c_1', m', p'\right\rangle_l, r', I, O'}{\langle c_1; c_2, m, p\rangle_l, r, I, O \Rightarrow \left\langle c_1'; c_2, m', p'\right\rangle_l, r', I, O'} \tag{2}$$

$$\frac{}{\langle \textbf{skip}; c, m, p\rangle_l, r, I, O \Rightarrow \langle c, m, p\rangle_l, r, I, O} \tag{3}$$

$$\frac{m(e) = \text{true} \qquad c = \textbf{while } e \textbf{ do } c_{\text{loop}}}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \left\langle c_{\text{loop}}; c, m, p\right\rangle_l, r, I, O} \tag{4}$$

$$\frac{m(e) = \text{false} \qquad c = \textbf{while } e \textbf{ do } c_{\text{loop}}}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m, p\rangle_l, r, I, O} \tag{5}$$

$$\frac{m(e) = v \qquad m' = m[x \mapsto v]}{\langle x := e, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m', p\rangle_l, r, I, O} \tag{6}$$

$$\frac{\begin{array}{c} c = \textbf{output } e \textbf{ to } o \qquad m(e) = v \\ \sigma_{\text{out}}(o) = l \qquad O' = \text{write}(O, o, v)\end{array}}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m, p\rangle_l, r, I, O'} \tag{7}$$

$$\frac{c = \textbf{output } e \textbf{ to } o \qquad \sigma_{\text{out}}(o) \neq l}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m, p\rangle_l, r, I, O} \tag{8}$$

$$\frac{\begin{array}{c} c = \textbf{input } x \textbf{ from } i \\ \sigma_{\text{in}}(i) \not\leq l \qquad m' = m[x \mapsto v_{default}]\end{array}}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m', p\rangle_l, r, I, O} \tag{9}$$

$$\frac{\begin{array}{c} c = \textbf{input } x \textbf{ from } i \qquad \sigma_{\text{in}}(i) = l \\ v = \text{read}(I, i, p) \qquad m' = m[x \mapsto v] \\ p' = p[i \mapsto p(i) + 1] \qquad r' = r[i \mapsto p'(i)]\end{array}}{\langle c, m, p\rangle_l, r, I, O \overset{\odot(i, p(i))}{\Rightarrow} \langle \textbf{skip}, m', p'\rangle_l, r', I, O} \tag{10}$$

$$\frac{c = \textbf{input } x \textbf{ from } i \qquad \sigma_{\text{in}}(i) < l \qquad r(i) \leq p(i)}{\langle c, m, p\rangle_l, r, I, O \overset{\otimes(i, p(i))}{\Rightarrow} \langle c, m, p\rangle_l, r, I, O} \tag{11}$$

$$\frac{\begin{array}{c} c = \textbf{input } x \textbf{ from } i \qquad \sigma_{\text{in}}(i) < l \qquad r(i) > p(i) \\ v = \text{read}(I, i, p) \qquad m' = m[x \mapsto v]\end{array}}{\langle c, m, p\rangle_l, r, I, O \Rightarrow \langle \textbf{skip}, m', p\rangle_l, r, I, O} \tag{12}$$

Figure 6. Local semantics for secure multi-execution.

$$\frac{\texttt{select}(L) = lec \quad lec, r, I, O \Rightarrow lec', r', I, O' \quad L' = L \setminus \{lec\} \cup \{lec'\}}{\langle L, wq, r, I, O\rangle \;\Rightarrow\; \langle L', wq, r', I, O'\rangle} \tag{1}$$

$$\frac{\texttt{select}(L) = lec = \langle \textbf{skip}, m, p\rangle_l \quad L' = L \setminus \{lec\}}{\langle L, wq, r, I, O\rangle \;\Rightarrow\; \langle L', wq, r, I, O\rangle} \tag{2}$$

$$\frac{\texttt{select}(L) = lec \quad lec, r, I, O \overset{\odot(i,n)}{\Rightarrow} lec', r', I, O' \quad wq(i,n) = L_w \quad L' = L \setminus \{lec\} \cup \{lec'\} \cup L_w \quad wq' = wq\,[(i,n) \mapsto \{\}]}{\langle L, wq, r, I, O\rangle \;\Rightarrow\; \langle L', wq', r', I, O'\rangle} \tag{3}$$

$$\frac{\texttt{select}(L) = lec \quad lec, r, I, O \overset{\otimes(i,n)}{\Rightarrow} lec', r', I, O' \quad wq(i,n) = L_w \quad wq' = wq\,[(i,n) \mapsto L_w \cup \{lec_i\}] \quad L' = L \setminus \{lec\}}{\langle L, wq, r, I, O\rangle \;\Rightarrow\; \langle L', wq', r', I, O'\rangle} \tag{4}$$

Figure 7. Global semantics for secure multi-execution.

on $\mathcal{L}$. The need for this scheduler corresponds to the intuition that we cannot make a low thread wait for the execution of steps in a high thread if we wish to avoid timing covert channels. We need the total order on $\mathcal{L}$ to make the scheduler function deterministic, which we require in some of our lemmas below.

Variables wq denote *waiting queues*. If a local execution state $\langle c, m, p\rangle_l \in wq(i,n)$ for some input channel $i \in C_{\text{in}}$ and $n \geq 0$, this means that the local execution state has tried to read from input channel i at position n, before the thread at security level $\sigma_{\text{in}}(i)$ has done. In such a case, the first thread will be placed in a waiting queue until the other thread executes the read operation. When that happens, the first thread will be executed further. The notation wq_0 denotes an initial waiting queue, mapping all pairs (i,n) onto the empty list.

Variables r in global execution states denote global input pointers. For any channel i, $r(i)$ is the position up to which the execution at security level $\sigma_{\text{in}}(i)$ has already read from channel i. We define r_0 as a synonym for the initial input channel pointer p_0, to be used when a global input pointer is meant.

We assume $\mathcal{L}$ to be finite and we define $L_{P,0}$, the initial set of local execution states for a program P, as the set of local execution states $\langle P, m_0, p_0\rangle_l$ for all $l \in \mathcal{L}$.

The program P is executed for input I by applying the global semantic rules from Figure 7 to the initial global execution state $\langle L_{P,0}, wq_0, r_0, I, O_0\rangle$. We write $\Rightarrow^*$ for the transitive and reflexive closure of the $\Rightarrow$ relation for global execution states. Suppose

$$\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \;\Rightarrow^*\; \langle L_f, wq_f, r_f, I, O_f\rangle \;,$$

with $\langle L_f, wq_f, r_f, I, O_f\rangle$ such that no semantic rule from Figure 7 applies, then we say that the secure multi-execution of program P with input channel state I produces final input pointer r_f and program output O_f, or $(P, I) \Rightarrow^* (r_f, O_f)$. From inspection of global and local semantics in Figures 7 and 6, we know that in such a case, L_f must be equal to the empty set $\emptyset$.

As we did for standard execution, we introduce a time-limited secure multi-execution execution relation $\Rightarrow^n$. We say that

$$\langle L, wq, r, I, O\rangle \Rightarrow^n \langle L', wq', r', I, O'\rangle$$

if $\langle L, wq, r, I, O\rangle$ can be transformed into $\langle L', wq', r', I, O'\rangle$ by n steps of the global execution semantics in Figure 7. If

$$\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \Rightarrow^n \langle L', wq', r', I, O'\rangle \;,$$

then we write $(P, I) \Rightarrow^n (r', O')$.

The following lemma identifies certain invariants on global execution states that the global semantics from Figure 7 preserve.

Lemma 1 (Global Execution State Invariants)**.** *Suppose that*

$$\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \;\Rightarrow^*\; \langle L, wq, r, I, O\rangle \;.$$

Then

- *for all $\langle c, m, p\rangle_l \in wq(i,n)$, we have that $l > \sigma_{\text{in}}(i)$, $p(i) = n$ and $n \geq r(i)$.*
- *if $\langle c, m, p\rangle_l$ is in L or $wq(i',n)$ for any (i',n), then $r(i) = p(i)$ for all $i \in C_{\text{in}}$ with $\sigma_{\text{in}}(i) = l$.*
- *for any security level l, there is only a single execution $\langle c, m, p\rangle_l$ at security level l in L or $wq(i,n)$ for any (i,n).*

Proof: It is clear that the results hold for the initial global execution state $\langle L_{P,0}, wq_0, r_0, I, O_0\rangle$ and is preserved by the global semantics from Figure 7. ■

IV. Theoretical Properties

A. Noninterference

We give both a regular definition of noninterference and a stronger one. Both definitions are termination-sensitive, but the strong definition is also timing-sensitive, while the normal one is not. We will show that secure multi-execution guarantees strong noninterference for any program P, and that it is precise for terminating runs of programs meeting the normal noninterference property.

For a given security level $l \in \mathcal{L}$, we define two program inputs I and I' to be equal up to l ($I =_l I'$) iff $I(i)$ equals $I'(i)$ for all $i \in C_{\text{in}}$ where $\sigma_{\text{in}}(i) \leq l$. Likewise, we define two program outputs O and O' to be equal up to l ($O =_l O'$) iff $O(o)$ equals $O'(o)$ for all $o \in C_{\text{out}}$, $\sigma_{\text{out}}(o) \leq l$. Finally, we define two input pointers p and

p' to be equal up to l ($p =_l p'$) iff $p(i) = p(i')$ for all $i \in C_{\text{in}}$, $\sigma_{\text{in}}(i) \leq l$.

The normal noninterference property does not take into account the timing covert channel. We formulate its definition in terms of an abstract transitive execution relation $\hookrightarrow^*$. Both the standard execution relation $\twoheadrightarrow^*$ and the secure multi-execution execution relation $\Rightarrow^*$ can be substituted for $\hookrightarrow^*$.

Definition 1 ((Normal) NonInterference)**.** *A program P is* timing-insensitively noninterferent *or simply* noninterferent *with relation to a given semantics $\hookrightarrow^*$ if for all security levels $l \in \mathcal{L}$ and for all inputs I and I' such that $I =_l I'$, we have that*

$$(P, I) \hookrightarrow^* \left(p_f, O_f\right)$$

if and only if

$$(P, I') \hookrightarrow^* \left(p'_f, O'_f\right)$$

and $p'_f =_l p_f$ and $O'_f =_l O_f$.

The strong definition takes into account both termination and timing covert channels. Again, we formulate this definition using a time-limited abstract transitive execution relation $\hookrightarrow^n$. Both the standard time-limited execution relation $\twoheadrightarrow^n$ and the time-limited secure multi-execution execution relation $\Rightarrow^n$ can be substituted for $\hookrightarrow^n$.

Definition 2 (Strong noninterference)**.** *A program P is* timing-sensitively noninterferent *or* strongly noninterferent *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 $O' =_l O$.

B. Soundness

The first of our two main results is the following soundness result.

Theorem 1 (Soundness of Secure Multi-Execution)**.** *Any program P is strongly noninterferent under secure multi-execution, using the* $\texttt{select}_{\text{lowprio}}$ *scheduler function.*

Because of space constraints, we prove only this result for the $\texttt{select}_{\text{lowprio}}$ scheduler. We do believe however that the same result would hold if we were to allow for the exceptions mentioned in Section II-D (I/O latency and independent progress on multi-core CPU's). In addition, we believe that a more relaxed form of noninterference (termination-sensitive noninterference) could be proven for a class of schedulers conforming to some basic fairness property.

We will need some lemmas in order to be able to prove this theorem.

Lemma 2 (Soundness Preservation for Local Semantics)**.** *Let l_g be a security level, and $l \leq l_g$. Suppose that*

$$\langle c, m, p\rangle_l, r_1, I_1, O_1 \overset{\sigma}{\Rightarrow} \langle c', m', p'\rangle_l, r'_1, I_1, O'_1 \ ,$$

where σ can denote a signal $\odot(i, n)$, $\otimes(i, n)$ or no signal at all.

Suppose that $r_2 =_{l_g} r_1$, $I_2 =_{l_g} I_1$ and $O_1 =_{l_g} O_2$. Then

$$\langle c, m, p\rangle_l, r_2, I_2, O_2 \overset{\sigma}{\Rightarrow} \langle c', m', p'\rangle_l, r'_2, I_2, O'_2 \ ,$$

with $r'_2 =_{l_g} r'_1$ and $O'_2 =_{l_g} O'_1$. In addition, if $\sigma = \odot(i, n)$, then $\sigma_{\text{in}}(i) = l \leq l_g$.

Proof: The steps are executed according to one of the rules from Figure 6. The result is clear from examining every rule. ■

Lemma 3 (Soundness Preservation for Local Semantics part two)**.** *Let l_g be a security level and $l \not\leq l_g$. Suppose that*

$$\langle c, m, p\rangle_l, r, I, O \overset{\sigma}{\Rightarrow} \langle c', m', p'\rangle_l, r', I, O' \ ,$$

where σ can denote a signal $\odot(i, n)$, $\otimes(i, n)$ or no signal at all. Then $r' =_{l_g} r$ and $O' =_{l_g} O$. In addition, if $\sigma = \odot(i, n)$, then $\sigma_{\text{in}}(i) = l \not\leq l_g$.

Proof: The execution step occurs according to one of the rules from Figure 6. The result is clear from examining every rule. ■

Before we continue, we need to define security-level limited equality for some additional concepts. We say that sets of local execution states L and L' are equal up to security level l_g (written $L =_{l_g} L'$) iff for all local execution states $lec = \langle P, m, p\rangle_l \in L$ with $l \leq l_g$, there is a local execution state $lec' \in L'$ such that $lec' = lec$ and vice versa. We say that waiting queues $wq =_{l_g} wq'$ iff $wq(i, n) =_{l_g} wq'(i, n)$ for all $n \geq 0$ and input channels i such that $\sigma_{\text{in}}(i) \leq l_g$.

Lemma 4 (Soundness Preservation for Global Semantics)**.** *Let l_g be a security level. Suppose that*

$$\langle L_1, wq_1, r_1, I_1, O_1\rangle \Rightarrow \langle L'_1, wq'_1, r'_1, I_1, O'_1\rangle \ ,$$

and that

$$\langle L_2, wq_2, r_2, I_2, O_2\rangle \Rightarrow \langle L'_2, wq'_2, r'_2, I_2, O'_2\rangle \ ,$$

with $L_1 =_{l_g} L_2$, $wq_1 =_{l_g} wq_2$, $r_1 =_{l_g} r_2$, $I_1 =_{l_g} I_2$ and $O_1 =_{l_g} O_2$. Suppose that the scheduler function $\texttt{select}_{\text{lowprio}}$ *is being used.*

Then we also have that $L'_1 =_{l_g} L'_2$, $wq'_1 =_{l_g} wq'_2$, $r'_1 =_{l_g} r'_2$, $O'_1 =_{l_g} O'_2$.

Proof: We define $lec_1 = \langle P_1, m_1, p_1 \rangle_{l_1} = \texttt{select}_{\text{lowprio}}(L_1)$ and we write $lec_2 = \langle P_2, m_2, p_2 \rangle_{l_2} = \texttt{select}_{\text{lowprio}}(L_2)$. We first handle the case that $l_1 \leq l_g$. Because $L_1 =_{l_g} L_2$, we have that $lec_1 = lec_2$. Inspection of the rules from Figure 7, together with Lemma 2 now easily yields the result.

In the alternate case $l_1 \not\leq l_g$, we know from Lemma 3 that $r'_1 =_{l_g} r_1 =_{l_g} r_2 =_{l_g} r'_2$, and $O'_1 =_{l_g} O_1 =_{l_g} O_2 =_{l_g} O'_2$. In addition, it is clear from inspecting the rules of the global semantics in Figure 7 that $L'_1 =_{l_g} L'_2$ and $wq'_1 =_{l_g} wq'_2$. ∎

We are now ready to give the proof of Theorem 1.

Proof of Theorem 1 (Soundness): Take a program P, any security level $l \in \mathcal{L}$ and two input channel states I and I', such that $I =_l I'$. Suppose that $(P, I) \Rightarrow^n (r, O)$, or

$$\langle L_{P,0},\, wq_0,\, r_0,\, I,\, O_0 \rangle \Rightarrow^n \langle L,\, wq,\, r,\, I,\, O \rangle$$

and $(P, I') \Rightarrow^n (r', O')$, or

$$\langle L_0,\, wq_0,\, r_0,\, I',\, O_0 \rangle \Rightarrow^n \langle L',\, wq',\, r',\, I',\, O' \rangle \ .$$

By induction on n, and using Lemma 4, we can easily prove that $L' =_l L$, $wq' =_l wq$, $r' =_l r$ and $O =_l O'$. ∎

C. Precision

Informally, we call a technique *transparent* for a program P with input I if it makes P produce the same externally observable results for this input. The *precision* of a technique is a measure of the set of pairs (P, I) for which the technique is transparent. More such pairs means higher precision. The second of our two main theorems gives a lower bound on the precision of secure multi-execution. We show that the technique is transparent for terminating runs of termination-sensitively noninterferent programs. This set of programs strictly includes all programs which are well typed under the type system described by Volpano and Smith [16].

Theorem 2 (Precision of Secure Multi-Execution)**.** *Suppose we have a noninterferent program P. Suppose that*

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

for some I, p and O. Then

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

Before we continue, we define the limitation of an input I to a security level $l \in \mathcal{L}$:

$$I|_l(i) = \begin{cases} I(i) & \text{if } \sigma_{\text{in}}(i) \leq l, \\ \_ \mapsto v_{default} & \text{otherwise.} \end{cases}$$

It is clear that $I|_l =_l I$.

Lemma 5 (Correspondence between Standard Execution and Secure Multi-Execution)**.** *Let $l \in \mathcal{L}$. Let P be a program. Suppose that*

$$\langle L_{P,0},\, wq_0,\, r_0,\, I,\, O_0 \rangle \Rightarrow^* \langle L,\, wq,\, r,\, I,\, O \rangle \ ,$$

with $\langle c, m, p \rangle_l \in L$. Define $I_l = I|_l$, then

$$\langle P,\, m_0,\, p_0,\, I_l,\, O_0 \rangle \rightarrow^* \langle c,\, m,\, p,\, I_l,\, O' \rangle \ ,$$

with $O'(o) = O(o)$ for all o such that $\sigma_{\text{out}}(o) = l$.

Furthermore, the number of global execution steps using rules (1)*, and* (3) *and involving a local execution step for a local execution state at security level l in the derivation of*

$$\langle L_{P,0},\, wq_0,\, r_0,\, I,\, O_0 \rangle \Rightarrow^* \langle L,\, wq,\, r,\, I,\, O \rangle$$

is equal to the number of standard execution steps in the derivation of

$$\langle P,\, m_0,\, p_0,\, I_l,\, O_0 \rangle \rightarrow^* \langle c,\, m,\, p,\, I_l,\, O' \rangle \ .$$

Proof: We present a proof by induction on the number of global execution steps used in the derivation of the statement

$$\langle L_{P,0},\, wq_0,\, r_0,\, I,\, O_0 \rangle \Rightarrow^* \langle L,\, wq,\, r,\, I,\, O \rangle \ .$$

If a global execution step was used with a local transition on a local execution state at security level $l' \neq l$, then the result is directly clear by inspection of the rules for the global and local semantics in Figure 7 and Figure 6.

For global execution steps from the list in Figure 7 involving a local execution step on a local execution state at security level l, we prove that we can make a corresponding step in the standard execution, maintaining the properties of this lemma.

For global semantic rule (1) from Figure 7 instantiated with one of the local semantic rules (1) through (6) from Figure 6, it is easy to verify that a corresponding standard execution step from Figure 5 applies to the standard execution state.

For both global semantic rule (1) instantiated with local semantic rule (12) or (9) and global semantic rule (3) with local semantic rule (10), standard semantic rule (8) can be applied to the standard execution state, ensuring the results of this theorem. For global semantic rule (4), with local semantic rule (11), the result is clear without an execution step on the standard execution state.

For global semantic rule (1) instantiated with local semantic rule (7) or (8), applying standard semantic rule (7) provides the correct result. ∎

We can now prove Theorem 2.

Proof of Theorem 2 (Precision): We know that

$$\langle P,\, m_0,\, p_0,\, I,\, O_0 \rangle \rightarrow^* \langle \textbf{skip},\, m_f,\, p,\, I,\, O \rangle \ .$$

1) Termination: Let $\langle L_{P,0}, wq_0, r_0, I, O_0\rangle$ be the initial global execution configuration for program P. We first prove that execution of $\langle L_{P,0}, wq_0, r_0, I, O_0\rangle$ terminates. Let's assume that it doesn't. Then there is an infinite list of global execution states

$$\begin{aligned}\langle L_{P,0}, wq_0, r_0, I, O_0\rangle &\Rrightarrow \langle L_1, wq_1, r_1, I, O_1\rangle \\ &\Rrightarrow \langle L_2, wq_2, r_2, I, O_2\rangle \Rrightarrow \cdots\end{aligned}$$

Because the sets of security levels for which there are local execution states in L_i are finite and descending, and because each global execution step applies a local execution step to one element of L_i there must be at least one security level l such that an infinite amount of global execution steps in this chain apply a local execution step to a local execution state at security level l. Clearly, at most one such execution step can be done using the global semantic rule (2), so because of Lemma 5, there must be an infinite set of standard execution states $\langle c_i, m_i, p_i, I_l, O_i\rangle$ such that

$$\begin{aligned}&\langle P, m_0, p_0, I_l, O_0\rangle \rightarrowtriangle^1 \langle c_1, m_1, p_1, l, I_l\rangle O_1 \\ &\langle P, m_0, p_0, I_l, O_0\rangle \rightarrowtriangle^2 \langle c_2, m_2, p_2, l, I_l\rangle O_2 \\ &\cdots\ ,\end{aligned}$$

with $I_l = I|_l$. However, we know that $(P, I) \rightarrowtriangle^* (p, O)$ and that P is noninterferent. Therefore, we also know that $(P, I_l) \rightarrowtriangle^* (p', O')$ with $p' =_l p$ and $O' =_l O$, and the standard execution of P with input I_l must terminate. Because standard execution is deterministic for our model language, this is a contradiction.

Now let $\langle L_f, wq_f, r_f, I, O_f\rangle$ be the global execution state such that

$$\langle L_{P,0}, wq_0, r_0, I, O_0\rangle \Rrightarrow^* \langle L_f, wq_f, r_f, I, O_f\rangle\ ,$$

and $\langle L_f, wq_f, r_f, I, O_f\rangle$ is terminated. It is clear that L_f is empty because otherwise one of the global execution rules would apply.

2) No eternal waiters: We prove by induction on the security levels $l \in \mathcal{L}$, ordered according to the total order on $\mathcal{L}$, that $wq_f(i, n)$ contains no execution states for security levels $\leq l$ for any pair (i, n).

Let l be a security level such that the induction hypothesis holds for all $l' < l$. Suppose that $wq_f(i, n)$ contains a local execution state $\left\langle c_{wq}, m_{wq}, p_{wq}\right\rangle_l$. Because of Lemma 1, we know that $\sigma_{\text{in}}(i) = l' < l$ and that $p_{wq}(i) = n$ and $c_{wq} \neq \textbf{skip}$. Because of Lemma 5, we then know that

$$\langle P, m_0, p_0, I_l, O_0\rangle \rightarrowtriangle^* \langle c_{wq}, m_{wq}, p_{wq}, I_l, O_{wq}\rangle\ ,$$

where $I_l = I|_l$. This last state cannot be stuck and because of noninterference of P, we know that P must terminate for the input I_l, so that

$$\begin{aligned}\langle P, m_0, p_0, I_l, O_0\rangle &\rightarrowtriangle^* \langle c_{wq}, m_{wq}, p_{wq}, I_l, O_{wq}\rangle \\ &\rightarrowtriangle^* \langle \textbf{skip}, m_{f,l}, p_{f,l}, I_l, O_{f,l}\rangle\ .\end{aligned}$$

Because c_{wq} must start with a statement of the form **input** x **from** i, we know that $p_{f,l}(i) > p_{wq}(i)$.

On the other hand, there is no local execution state at security level l' in L_f or $wq(i, n)$ for any (i, n). Therefore, global execution rule (2) from Figure 7 must have been applied to a local execution state of the form $\langle \textbf{skip}, m', p'\rangle_{l'}$. But then again Lemma 5 tells us that

$$\langle P, m_0, p_0, I_{l'}, O_0\rangle \rightarrowtriangle^* \langle \textbf{skip}, m', p', I_{l'}, O'\rangle\ ,$$

with $I_{l'} = I|_{l'}$. Because of noninterference and because $I_{l'} =_{l'} I_l$ and $I =_{l'} I_l$, we know that $p' =_{l'} p$ and $p =_l p_{f,l}$, so $p'(i) = p(i) = p_{f,l}(i) > p_{wq}(i)$. However, Lemma 1 tells us that $p'(i) = r_f(i)$ and $r_f(i) \leq p_{wq}(i)$, which is a contradiction.

3) Correct I/O: All that remains to prove is that $r_f = p$ and $O_f = O$. Because L_f and $wq(i, n)$ are empty for all (i, n), global semantic rule (2) must have been applied to an execution of the form $\langle \textbf{skip}, m_l, p_l\rangle_l$ for any level $l \in \mathcal{L}$. Lemma 5 then tells us that

$$\langle P, m_0, p_0, I_l, O_0\rangle \rightarrowtriangle^* \langle \textbf{skip}, m_l, p_l, I_l, O_l\rangle\ ,$$

where $I_l = I|_l$ and $O_l(o) = O_f(o)$ for all o such that $\sigma_{\text{out}}(o) = l$. Because P is noninterferent and $I_l =_l I$, we have that $O_l =_l O$ and $p_l =_l p$. We know from Lemma 1 that $r_f(i) = p_l(i) = p(i)$ for all i such that $\sigma_{\text{in}}(i) = l$. Because the above holds for any l, the theorem is proven. ∎

V. Experimental Behaviour

In order to get a better understanding of the practical behaviour of secure multi-execution, we performed an experiment. We implemented a model browser using the Mozilla Spidermonkey Javascript engine [10], and tested it on the set of benchmarks used by the developers of another major Javascript engine (Google Chrome V8) [11]. In this section, we discuss the results.

A. A model browser using secure multi-execution

Implementing secure multi-execution in a model browser does not require any modifications to the Javascript engine internals. Instead, several instances of the engine need to be constructed, and all input/output operations need to be modified according to the appropriate rules from Figure 6 and 7.

We have done this exercise using the Mozilla Spidermonkey Javascript engine. We exposed model input/output functions on two security levels ("hi" and "lo"). We have implemented the modified secure multi-execution semantics for I/O operations using the Mozilla NSPR library of concurrency primitives. Each I/O function contains an artificial 10 milli-seconds "sleep" call, simulating I/O latency. Two scheduling

```
for (var i = 0; i < 100; ++i) {
  var test = 0;
  for (var j = 0; j < 10000; ++j) {
    test += j;
  }
  if (i % 10 == 0) {
    var hi_in = hi_input();
    var lo_in = lo_input();
    lo_output("#" + i + ". lo_in: '"
      + lo_in + "'. hi_in is: '"
      + hi_in + "'");
    hi_output("#" + i + ". hi_in: '"
      + hi_in + "'. lo_in is: '"
      + lo_in +"'");
  }
}
```

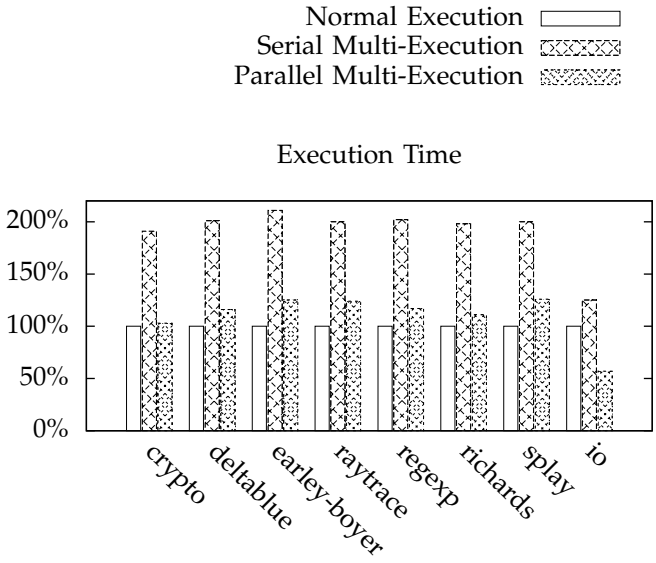
Figure 8. Source code for the additional IO benchmark (not including administrative benchmark-related code).

strategies were implemented, one implementing a simple serial scheduling of the different executions, starting with the lo one. The other scheduling strategy executed both executions in parallel, while indicating to the OS that the "lo" execution should be prioritised. For comparison, we have also built a standard Javascript engine, implementing the I/O functions in a standard way, and only executing a Javascript program a single time. All concurrency operations were removed in this last implementation.

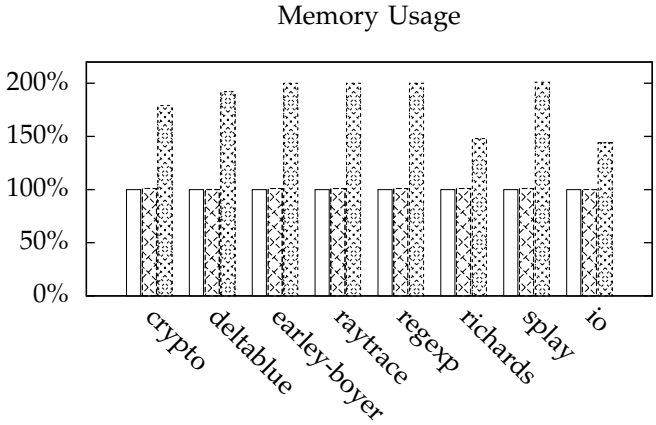
B. An Experiment

For our experiment, we have worked with the Google Chrome V8 Benchmark suite. This suite is constructed by the developers of the Google Chrome V8 Javascript engine in order to compare their engine to others. The suite contains a set of 7 benchmarks (Richards, DeltaBlue, Crypto, RayTrace, EarleyBoyer, RegExp, Splay), some explicitly focusing on certain components of the Javascript engine (such as the garbage collector in Splay or the regular expression engine in RegExp), some more general. These benchmarks were modified in two ways. First, we removed calls to the load function, instead concatenating all relevant files in a single file. Second, we modified the benchmarking code to execute each benchmark 5 times, instead of stopping after 1 second has passed.

Furthermore, we have added a benchmark called IO, simulating a program performing a large amount of I/O operations on channels at different security levels. The source code for this benchmark is shown in Figure 8.



(a) Execution Time (Percentage of normal).



(b) Memory Usage (Percentage of normal).

Figure 9. Experimental Results.

C. Results

In Figure 9, we show graphs of our measurements for execution time and memory usage of the three Javascript executors described above, for each of the benchmarks. The tests were run on a recent dual-core computer with a sufficient amount of RAM [1]. We define execution time as elapsed wall clock time as measured by the Ubuntu 7.10 time utility. Memory usage was measured in a separate run of the programs instrumented using the Valgrind Massif tool [17], and is defined as the amount of heap memory allocated through malloc and similar functions.

The results allow us to make certain important observations, both for the parallel and serial multi-

[1]CPU: Intel Core 2 Duo P8600 2.4 GHz, Amount of RAM: 2GB.

execution. Serial multi-execution has only a limited impact on memory usage. This could have been expected, because the high security level execution is only started when the low security one has completely terminated. On the other hand, serial multi-execution increases execution time by a factor of two or more. This is clearly also what would be expected for this situation. For the I/O benchmark, the added execution time is limited, thanks to a phenomenon we discuss further below.

For parallel multi-execution, the situation is more or less reversed with respect to serial multi-execution. Memory usage increases by a factor between 50% and 100%. Two instances of the program are continuously running in parallel, each with their own copy of the internal state of the program. The increase in execution time is rather limited, ranging between 2% and 25%. Both executions execute in parallel on the two cores of the CPU.

When we first ran these benchmarks, we had not anticipated the surprisingly good results for the IO benchmark. It is counterintuitive that executing a program twice can actually decrease total execution time. This phenomenon can be attributed to a better handling of I/O latency by the parallel runs of the program (recall that we have simulated I/O latency by adding a 10ms sleep in each I/O operation). For the example in Figure 8, one should imagine that a single program performing all four I/O operations serially is replaced by two programs, each performing only the two I/O operations on its security level, and progressing in parallel on separate cores of the CPU (reusing an I/O result from a lower execution does not produce any latency). For this benchmark, secure multi-execution in effect performs an automatic parallelisation, based on the assumption of noninterference. Any application performing I/O operations at different security levels serially, will benefit from this effect, although it will of course be more limited than in this artificial benchmark.

D. Discussion

We think the results in Figure 9 show that secure multi-execution is not only of theoretical significance, but is also viable in real-life browser environments. Especially parallel multi-execution seems to provide a good compromise on modern consumer computer systems where memory is often no longer the bottleneck and multi-core CPUs are common. We think these results indicate an interesting direction for further research.

Note also that we did not perform any optimisation in our benchmarks. One technique which looks promising in this respect is program slicing [18]. An execution of a program at a security level l could be optimised by slicing the program for a slicing criterion consisting of all input and output statements at security level l, and the variables they depend on. Depending on the structure of the dependency relations in the program, this could lead to a considerable increase in efficiency.

The source code for this experiment is available online [19].

VI. Discussion

A. Secure Multi-Execution in a Real Browser

In Section V we discussed how the technique can be implemented in a model browser. In order to implement it in a real browser, support needs to be added for some additional concepts. In this section, we discuss some such concepts, and outline some proposed solutions for supporting them. We do not provide full details and further research into this topic is needed.

One concept we have not modelled in our formal presentation are browser callbacks. At certain moments in time, the browser will call certain Javascript functions to allow the program to react to certain events. For example, "document.onload" event handlers will be called after the HTML page has finished loading, to allow Javascript programs to perform some initialisation. Browser callbacks can be encoded in our model language by adding an event loop at the end of a program P, looping on an input from a $I_{NextEvent}$ input channel. On this input channel, the default value $v_{default}$ would correspond to an empty event. By placing event sources from more than one channel in the event loop, a policy could be encoded where not all callbacks are associated to the same security level.

This formal encoding of browser callbacks corresponds to an implementation where callbacks are executed only in executions at security level l or higher, with l the security level assigned to the relevant event. Typically, a browser's "document.onload" event would be associated to a lower security level than an "element.onkeypress" event. Note finally, that by also modelling an event when leaving the page, event-driven web applications can still be considered as terminating.

Another important feature to support is the browser's DOM tree. The DOM tree can be described as the complete set of interfaces which a browser exposes to Javascript programs. It is a tree structure consisting of DOM nodes. Certain DOM nodes are intended to be called as functions, or to be read from and assigned as variables. Some represent parts of the current HTML document. DOM nodes can also be added or deleted etc. Nodes have names and can be

referred to using top-down paths of the form "document.body.firstDiv".

For implementing support for the DOM tree in an implementation of secure multi-execution, we need to distinguish different types of interaction with the DOM tree. First of all, certain function calls and DOM node property assignments will trigger side effects in the outside world. For example, setting the URL of an HTML `img` element may cause an HTTP request to be made to get the new image. Adding an HTML `div` element with some text to document.body will add text in the user-visible document. These types of interactions have to be modelled using a separate input channel for every possible list of arguments to the function or assignment. It would be useful to support that for a single node, not all of these input channels are assigned the same security level. For example, when setting the URL of a HTML `img` element, the standard *same origin policy* (see e.g. Johns [12]) could be used to determine whether the assigned URL is trusted or not.

Interactions with the DOM tree not producing external side effects can be implemented in one of two ways. The distinction comes down to the question whether the DOM tree is modelled as part of a Javascript program's state or as part of the outside world. The latter alternative means mapping every possible interaction with the DOM tree to an input channel, with security level corresponding to the security level of the DOM tree nodes and their values. This is a viable option, but probably introduces quite some overhead for programs interacting heavily with the DOM tree because of the bookkeeping associated with input handling in secure multi-execution. In addition, we think this choice will result in a lower precision technique than the alternative.

The alternative solution is to incorporate the DOM tree into each Javascript execution's state, in effect giving each execution its own copy of the DOM tree, as is done for program variables. Different executions will not see identical copies of the DOM tree, but instead an evaluation of the DOM tree in terms of security levels could be used to decide what each exposed DOM tree should look like. One could imagine a situation where all security levels except the highest only get to see a dummy version of the HTML document. In mash-up applications, one could even imagine having different security levels each seeing only separate parts of the real DOM tree. Care should be taken to align this approach with the handling of DOM tree nodes producing external side effects.

We suspect the latter alternative is preferable, even though it adds a certain memory overhead for the additional copies of the DOM tree. This additional memory overhead can be argued to be similar to the general overhead of secure multi-execution for objects manipulated by a program. In addition, if necessary, we think optimisations can be imagined like a copy-on-write-based sharing of the tree etc.

In this section, we have described some of the challenges we anticipate in a real-life browser-based implementation of secure multi-execution. We think there is interesting further research to be done in this direction.

B. What about Non-Noninterferent Programs?

In Section IV-C, we have proven that termination-sensitively noninterferent programs produce the same results as they do under normal execution, if the normal execution terminates for a given input. An important question is then what results secure multi-execution produces for interferent programs. Theorem 1 in Section IV-B tells us that any program is noninterferent under secure multi-execution. This means that multi-execution has some way of replacing interferent behaviour by noninterferent behaviour. In this section, we explore how it achieves this for some example programs.

Secure multi-execution in fact has different ways to block unwanted information flows. The examples in the informal overview in Section II already show how explicit and implicit information flows are handled. The offending assignments and conditional statements will still be executed, but any data they don't have access to will be replaced by default data. In the high security executions, the offending statements will be executed with the real data, but no leaks to low security output channels will be possible.

Another mechanism in effect blocking certain types of covert flows can be observed for the example in Figure 10a. We assume that reading a value from L can produce external side effects. Under standard execution, this program will read from L iff $x == v_{default}$ and is thus clearly interferent. Under secure multi-execution, the low execution will not read any input from L. The high execution will try to read from L iff $x == v_{default}$, but if it does, it will block and wait indefinitely for the low execution to read from L. This behaviour is termination- and timing-sensitively noninterferent. Secure multi-execution blocks the undesirable flow by making the high security thread wait indefinitely in the global waiting queue. Figure 10b shows the same phenomenon for a termination-insensitively noninterferent program.

Figure 10c shows a nonterminating termination-sensitively noninterferent program, for which multi-execution is not transparent using the $\texttt{select}_{\text{lowprio}}$

input x **from** H
if $x == v_{default}$ **then skip else input** x **from** L

(a) An interferent program.

input x **from** H
if $x == v_{default}$ **then while** true **do skip else skip**
input x **from** L

(b) A termination-insensitively noninterferent program.

output 1 **to** H
while true **do skip**;

(c) A termination-sensitively noninterferent program.

Figure 10. Example programs highlighting some interesting behaviour under multi-execution.

scheduler. In this example, the high security execution will never be selected because the low-security thread diverges. This is an unfortunate effect, which can be mitigated by using a more relaxed scheduler. An interesting question is whether this is possible without compromising timing-sensitivity in the soundness guarantee.

The fact that secure multi-execution can enforce noninterference for any program makes it well suited for situations where one has little control over the code to be executed, for example in a web browser. When such code is interferent, secure multi-execution will still execute it, but will modify its behaviour to be non-interferent. For this to work well, it is important that the modified behaviour remains as meaningful as possible. We believe that in many situations, the modifications that secure multi-execution makes are as meaningful as possible without compromising noninterference. For example, a web application which accidentally sends private information to a website statistics service, will be modified to send a request based on the default values replacing the private data in the low execution. We consider this to be desirable behaviour in a situation where transparency conflicts with noninterference.

In some situations, secure multi-execution is not able to produce such meaningful behaviour, e.g. when finishing execution with non-empty waiting queues. In other situations, secure multi-execution can detect that modifications to the behaviour have been made, e.g. when a H execution would have sent different data to public outputs than the L execution. In these situation, an implementation could issue an appropriate warning to the user, explaining that the original program was probably interferent and what kind of effects should be expected. To support this, we think it could be interesting subsequent work to perform a detailed formal analysis of the results that can be expected for programs conforming to different noninterference criteria.

C. Exceptions, Concurrency and Nondeterminism

Exceptions, concurrency and nondeterminism are programming language features which pose additional challenges for various techniques enforcing noninterference. First of all, in the context of noninterference, concurrency and nondeterminism are difficult to work with from a theoretical perspective. Noninterference then becomes a property one can no longer expect to enforce, and trickier concepts like possibilistic or probabilistic noninterference [20] are needed to formulate the desired properties. In addition to this additional challenge for researchers, exceptions and concurrency also poses additional challenges for the techniques themselves. For example, Smith and Volpano need to impose considerable extra restrictions in their type system to handle exceptions [16] and concurrency [21].

An appealing feature of secure multi-execution, is that there is no fundamental obstacle for it to support exceptions, concurrency or nondeterminism. The basic idea of executing the program once on every security level and handling input and output in each execution as described above, remains valid if we let each separate execution handle exceptions, concurrency and nondeterminism internally. It is intuitively still clear that no information on higher levels can leak to an execution on a lower level, because it simply does not get access to the information, and has no way of communicating with executions on higher levels. Nevertheless, developing these ideas into a formal proof remains a considerable challenge.

VII. Related Work

There is a vast amount of related work in the research area of information flow security. We point the reader to the excellent survey by Sabelfeld and Myers [20] for an overview of static techniques, and to the PhD thesis of Le Guernic [22] for an overview of dynamic techniques. In this related work section, we limit our discussion to papers that propose ideas or approaches closely related to our notion of secure multi-execution, and to work that applies information flow analysis to Javascript.

In order to close internal timing channels [15], Russo et al. describe an approach based on program transformation. For any conditional branching on high security variables, they rewrite the program to execute the

branches in dedicated threads. In these high security threads, they replace low security variables by high security images (copies) and implement careful synchronisation to avoid introducing data races. We believe their technique can in fact be seen as a sort of multi-execution. Their base thread is the equivalent of a low security thread, while the high threads they launch are in fact serialised by the added synchronisation commands to the equivalent of a single high thread. The difference comes down to the fact that instead of executing the full program on the low security level with fake high data, their equivalent of the low thread skips assignments to and branches on high security variables and forbids the use of high security variables. Because of this, they do not need to duplicate low variables or parts of the program not using high variables. On the results side, secure multi-execution seems to provide stronger guarantees (timing- and termination-sensitive noninterference vs. termination-insensitive noninterference). Russo et al. do not prove any form of precision, but our impression is that secure multi-execution is more precise than their transformation technique.

Pottier and Simonet [23] discuss an approach to prove noninterference using standard preservation and progress theorems for a security type system for Core ML. Since noninterference is not a safety property (one needs to consider two executions to reason about noninterference), they propose adding a pairing construct to the language that makes it easy to reason about two executions of a program. This pairing construct is somewhat similar to bi-execution in our approach. However, Pottier and Simonet use it as a theoretical construct to reduce noninterference to subject reduction for an extended programming language.

Somewhat similarly, Barthe et al. [24] propose the use of *self-composition*, another pairing construct, to reason about noninterference in program logics. Their goal is to support the use of (Hoare-logic like) program verification techniques to verify noninterference properties.

Vogt et al. [25] describe a practical taint tracking technique for Javascript. Their main goal is providing reliable and efficient protection against cross-site-scripting attacks. While practical and useful, their technique is not fully sound: Russo et al. [7] identify a number of issues where the technique by Vogt et al. is unsound. They go on to propose a provably sound execution monitor for tracking information flow in DOM-tree like data structures, but this monitor has not yet been implemented. The only existing approach for tracking information flow in Javascript that is provably sound and has been implemented for full Javascript is a technique recently proposed by Chugh et al. [13]. They propose a framework for staging information flow intended to handle dynamically generated Javascript. They limit attention to specific types of flow policies such that the residual checks that the browser needs to perform can be efficient. They also propose a static instantiation of the framework using an analysis technique based on set inclusion constraints. As any static technique, this technique is not fully precise and will reject programs for which secure multi-execution can enforce noninterference transparently.

Yumerefendi et al. [26] describe a comprehensive information flow control system called *TightLip*, implemented in the Linux kernel. In order to detect insecure information flows, TightLip will spawn a *doppelganger* (look-alike) process in parallel to processes accessing confidential files. The doppelganger process inherits most of the state of the original, but is only given access to a *scrubbed* (having all confidential data removed) version of the file. All subsequent system calls of both processes are then tracked and compared to each other. If all of these system calls are identical, the process is assumed to be noninterferent and its execution is left unmodified. If a difference is detected, TightLip's policy module decides whether to block the system call, kill the process, scrub output buffers, replace the original process by its doppelganger, transitively mark affected files or pipes as sensitive or do nothing.

TightLip and secure multi-execution share the idea of executing a process multiple times on different security levels and replacing high security input in the low-security execution. However, Yumerefendi et al. use this idea only to detect divergence of outputs and thus information leaks. Because it is their original process which produces all output, and because one can no longer meaningfully compare system calls after the first difference, they have to treat all further output as sensitive, and resort to relatively crude mitigation techniques. They do not provide any formal results and as they seem to recognize, their transitive marking of files affected by system calls as sensitive is unsound because of improper flow-sensitivity [27].

VIII. Conclusion

We have proposed *secure multi-execution*, a novel dynamic enforcement mechanism for noninterference policies. Secure multi-execution enjoys interesting theoretical properties. We have shown that it is sound for a very strong notion of noninterference taking into account the termination and timing covert channels, and that it is precise in the sense that the enforcement is transparent for all terminating runs of all termination-sensitively noninterferent programs. We have also provided evidence that secure multi-execution can be

practical by reporting benchmark results on an implementation of the technique for Javascript.

IX. Acknowledgements

The idea of secure multi-execution grew from an interesting question that Nicky Mouha raised in a summer school lecture on information flow security. Nicky: we hope this paper finally provides a satisfactory answer to your question!

We are grateful to Andrei Sabelfeld, Bart Jacobs and Dave Clarke for interesting feedback and comments on draft versions of this paper. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

[1] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of computer security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[2] N. Heintze and J. G. Riecke, "The SLam calculus: programming with secrecy and integrity," in *POPL*, 1998, pp. 365–377.

[3] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *POPL*, 1999, pp. 228–241.

[4] M. Zanotti, "Security typings by abstract interpretation," in *Proc. Symposium on Static Analysis*, 2002, pp. 360–375.

[5] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Comm. of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[6] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt, "Automata-based confidentiality monitoring," in *ASIAN*, 2006, pp. 75 – 89.

[7] A. Russo, A. Sabelfeld, and A. Chudnov, "Tracking information flow in dynamic tree structures," in *ESORICS*, 2009, pp. 86–103.

[8] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: An architectural framework for user-centric information-flow security," in *MICRO*, 2004, pp. 243–254.

[9] F. B. Schneider, "Enforceable security policies," *ACM Trans. Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[10] "Mozilla spidermonkey website." [Online]. Available: http://www.mozilla.org/js/spidermonkey/

[11] "Google chrome v8 benchmark suite instructions." [Online]. Available: http://code.google.com/apis/v8/benchmarks.html

[12] M. Johns, "On JavaScript malware and related threats," *Journal in Computer Virology*, vol. 4, no. 3, pp. 161–178, 2008.

[13] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for javascript," in *PLDI*, 2009, pp. 50–62.

[14] S. Maffeis, J. C. Mitchell, and A. Taly, "Isolating javascript with filters, rewriting, and wrappers," in *ESORICS*, 2009, pp. 505–522.

[15] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld, "Closing internal timing channels by transformation," in *ASIAN*, 2006, pp. 120–135.

[16] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Computer Security Foundations Workshop*, 1997, pp. 156–168.

[17] "Valgrind user manual - massif: a heap profiler." [Online]. Available: http://valgrind.org/docs/manual/ms-manual.html

[18] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[19] D. Devriese and F. Piessens, "Secure multi-execution experiment source code." [Online]. Available: http://www.cs.kuleuven.be/~dominiqu/permanent/sme-experiment.tar.gz

[20] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[21] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL*, 1998, pp. 355–364.

[22] G. Le Guernic, "Confidentiality enforcement using dynamic information flow analyses," Ph.D. dissertation, Kansas State University, 2007.

[23] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 117–158, 2003.

[24] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *CSFW*, 2004, pp. 100–114.

[25] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *NDSS*, 2007.

[26] A. R. Yumerefendi, B. Mickle, and L. P. Cox, "TightLip: Keeping applications from spilling the beans," in *NSDI*, 2007.

[27] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," 2010, unpublished.