Information Flow Monitor Inlining

Andrey Chudnov David A. Naumann
Stevens Institute of Technology
Hoboken, NJ, USA
{achudnov,dnaumann}@stevens.edu

***Abstract*—In recent years it has been shown that dynamic monitoring can be used to soundly enforce information flow policies. For programs distributed in source or bytecode form, the use of just-in-time (JIT) compilation makes it difficult to implement monitoring by modifying the language runtime system. An inliner avoids this problem and also serves to provide monitoring for more than one runtime. We show how to inline an information flow monitor, specifically a flow sensitive one previously proved to enforce termination insensitive noninterference. We prove that the inlined version is observationally equivalent to the original.**

***Keywords*-Information flow, information security, reference monitors, program transformation**

I. Introduction

Many security requirements involve information flow. Many practical security measures involve runtime mechanisms — but it was long thought that information flow requires static enforcement mechanisms because it is not a trace property [29], [9] and runtime monitoring enforces trace properties [35]. Hamlen et al [22] show that any property that can be statically analyzed can also be checked by a runtime monitor with access to the program source. Le Guernic et al. [20] confirm this for an information flow monitor that performs some static analysis. They suggest that their security automaton could be implemented by program transformation. Subsequent work has explored dynamic information flow monitoring, but not by transformation. In this paper, we argue that, for web applications, inlining is the only way to go, and we show how to prove an inlining transformation correct.

Security policy assigns categories ("levels", related by a partial order $\sqsubseteq$ [12]) to input and output channels with the intention that information may flow from an input of level l to an output of level l' only if $l \sqsubseteq l'$. The notion of "information" is formalized by non-interference [10], [17]: in multiple runs of the program, variation in an input at level l is allowed to be correlated with variation of an output at level l' only if $l \sqsubseteq l'$.

Monitoring of the flow of information via data flows is called taint tracking; it has been implemented for machine code [44], bytecode [30], and interpreted source languages such as JavaScript [13], [31]. Taint tracking is quite satisfactory in some circumstances (but see [24]). It is inherently incomplete because information easily flows via combinations of control and data flows (usually called "implicit" and "explicit" information flow). In this paper we are concerned with implicit as well as explicit flows, but not covert channels.

Runtime monitoring has the potential to be more permissive than purely static enforcement, in the sense that it can allow secure executions even if the program also has insecure ones which must not be allowed to run to completion. And the determination of whether a run is secure can be made using precise runtime information rather than just the conservative approximations used in static analysis.

One particularly important scenario in which information flow control is needed is web applications, especially in the browser which interleaves execution of many programs from sources accorded widely varying degrees of trust. The work in this paper is intended to contribute towards the tracking of implicit and explicit information flows in popular web languages such as JavaScript. These languages are dauntingly complex and unattractive for static analysis. Worse yet, the ubiquitous use of dynamic code generation (i.e. **eval**) is a serious impediment to static analysis (though it is not impossible [8]). In theory, runtime monitoring is better suited to handle **eval** [3].

Runtime monitoring for an interpreted language like JavaScript or Java is in principle easier than monitoring at the hardware level: complete mediation can be achieved by modifications of the language runtime system since it is involved with every control and data flow event. Runtime monitors are usually implemented as a part of the virtual machine (VM) for a given language. However, we find such VM-based monitoring impractical in the browser and some other settings whereas inlining of a monitor appears to be practical. Below we give three reasons for it.

First of all, there are quite a few JavaScript VMs in wide deployment, e.g. SpiderMonkey and TraceMonkey by Mozilla [16], V8 by Google [18] and SquirrelFish by Apple. Internally they are very different, so the implementation of a monitor for one runtime would not be useful for the others. Moreover, the rapid pace of browser development has rendered prototype monitors obsolete before they were completed. By contrast, an inliner can be portable and it can create portable code.

The second argument in favor of inlining has to do with the fact that modern run-times feature *just-in-time* (JIT)

compilation. JIT compilation is a process for generating native machine code for a program at run-time. Before it was introduced, the programs in languages like Java and JavaScript were interpreted: either directly, or after being compiled to an intermediate *bytecode* representation. This gives us portability, but we lose performance. JIT compilation allows to regain the performance by converting parts of the program to native code. One can't convert all the code at once, because this would slow down program start-up. Instead, the modern runtimes either convert the code method-by-method upon the first invocation (e.g. Microsoft .NET [33] and Google V8 [18]), or, in a more elaborate fashion, compile only the bodies of the loops where the interpreter spends most of the time (e.g. Mozilla TraceMonkey [16] and Sun HotSpot [38]). The key point is that the parts of the program translated to native code are no longer under the step-by-step control of the VM: It is no longer able to mediate every data and control flow event, which we need for information flow (as opposed, e.g., to monitoring system calls for access control).

A solution is evidently for monitoring to be inlined into the compiled code. This could be done in connection with a VM monitor, though adding further complication to what are already complicated interpreter/compilers. Inlining may be done at the level of source code or bytecode, so that one inliner can serve as the front end to many VMs, benefiting from code optimizations of the JIT compiler as well as other performance advances. JavaScript code is distributed in source form, and there is no standardized bytecode, so inlining should be done on source.

The third reason to do inlining is the potential to optimize the monitoring code. This was the primary motivation behind the previous work on access control monitor inlining [14], [15] which dates more than a decade back. Access control monitors are, however, significantly different from those proposed for information flow. And, the optimization technique that is most beneficial for information flow monitors is constant folding.

As far as we know, ours is the first work to prove correctness of an inlined information flow monitor. There are two key properties: *security*, in the sense that allowed runs satisfy termination-insensitive noninterference, and *transparency*, which means the monitor may halt insecure runs but does not otherwise change the observable behavior of the program. Instead of proving these properties directly, we base our work on a VM monitor which serves as specification and which has been proved to be secure [34]. It suffices to prove that the observable behaviors of our transformed programs are the same as those run under the VM monitor. This is tricky to get right. In this paper we present a modular way to specify inlining of a VM monitor, a way to organize the proof of correctness of this process, and carry out such proof.

In Sect. II we discuss related work and we indicate why we expect our proof technique to be useful for other source languages and monitoring schemes. In Sect. III we review the programming language and VM monitor studied by Russo and Sabelfeld [34], including its security property. Sect. IV presents our inlining transformation; Sect. V states the correctness results and sketches the proofs. Sect. VI discusses the prospects for practical inlining. Full details of our proofs can be found in the extended version on the authors' home pages.

II. Related work

A. Information flow monitoring

The problem of monitoring information flow has received wide attention recently, especially in the context of JavaScript security. From the practical side there have been developed at least two modifications of Mozilla's Firefox browser that introduce information flow monitoring in the JavaScript engine [31], [13]. Another information flow monitor implementation was done for Java VM [30]. Other taint tracking work was cited earlier.

Our interest is in provably sound tracking that handles implicit flows, which has only recently been achieved [40], [20], [36]. Shroff et al. [36] handle implicit flows via function pointers as well as explicit branches, which makes control flow tracking nontrivial. An interesting feature of their work is that control dependencies between program points may be learned over the course of multiple runs (e.g. test runs), eventually yielding soundness without the cost and/or conservativity of full static analysis.

The monitor of LeGuernic et al. [20] is flow sensitive which means the security level of a memory location may vary from one program point to another; it has been extended to concurrency [19]. Flow sensitivity is essential for low level code (due to reuse of registers and other memory locations). It is also essential for dynamic monitoring of code that isn't equipped with security annotations — whereas, for static analysis, label inference and label polymorphism may lessen the need for flow sensitivity.

Russo and Sabelfeld [34] have recently studied the characteristics of purely dynamic and hybrid (i.e., dynamic and static) information flow monitors, in connection with flow sensitivity. In the course of their work, they adapt the monitor of [20], reformulating it in a modular way using labeled transitions and proving a stronger security property that takes intermediate outputs into account. This is the basis for our work, so we discuss it at length in Sect. III.

The "no sensitive upgrade" rule for dynamic monitoring [42], [4] avoids the need for static analysis to determine implicit flows due to branches not taken. But it can reject secure computations. We and others are currently investigating its practical applicability. If it is not found to reject too many useful programs, "no sensitive upgrade" would be a very convenient way of implementing the monitor since it simplifies runtime monitoring and avoids the high cost and conservativeness of static analysis in the presence of

aliasing and **eval**. Absent definitive evidence about practicality, we believe it is important to explore the feasibility of inlining the monitors that rely on static analysis, for which transparency and other properties are more challenging.

B. Monitor in-lining

Zhu et al. [44] implement taint tracking by transforming binary code, focusing on engineering issues and experimentation but not proving correctness (nor dealing with implicit flow). Taint tracking at the level of operating systems objects also has a long history and recent resurgence, e.g. [43].

There has been extensive work on inlined monitoring for security policies that concern control flow —in particular, calls on sensitive APIs— as opposed to control flow as an information channel. Being pioneered by Erlingsson and Schneider for Java in [14], it was later applied to other languages and run-times. Yu et al. described a similar monitor for JavaScript in [41]. For assurance that the inlined program satisfies the security property, Hamlen et al. [21] use a type system to certify in-lined monitors for the .NET run-time. Sridhar and Hamlen [37] use model checking to certify in-lined monitors for ActionScript bytecode.

Besides security, it is desirable that monitoring is transparent; this is proved in some recent works including [1], [39]. These works are impressive in covering substantial fragments of JVM/CLR bytecode. Dam et al. [11] prove security and transparency for a language with concurrency, unlike our work and the others we cite on information flow monitoring.

Theoretical studies have formulated monitoring in terms of arbitrary state machines [35]. But the practical inlining systems cited above are for policies about control flow events; monitoring is needed only for invocations of security-sensitive methods. These systems are not easily adapted to track intermingled implicit and explicit flows of information. For example, although Le Guernic et al. [20] explicitly formulate their monitor as an automaton, their results are proved from scratch rather than drawing on general results about security automata.

Although they were focused on solving a slightly different problem, Chugh et al. in [8] describe semantics of a taint-tracking monitor for a significant subset of JavaScript as a set of code rewrite rules, which could be seen as a monitor inlining process.

After we completed the work reported here, we became aware that Magazinius et al. [28] have independently devised an inlined monitor for information flow. They treat the simple imperative language without output commands but with **eval**, which requires dynamic inlining, i.e., runtime transformation of each string value to which **eval** is applied. To this end, the language includes procedure definitions so that the transformation can itself be included in the program. The complication of static analysis is avoided by using the the "no sensitive upgrade" rule (see Sect. II-A). Some performance experiments are reported. A pre-post noninterference theorem is stated and presumably the transparency property holds.

Our inlining transformation is syntax-directed (as is the one in in [28]). In the absence of output commands, it might be convenient to prove transparency and/or security using a compositional semantics, perhaps via Benton's technique [6]. However, with intermediate outputs a compositional semantics is relatively complicated. In any case recent works including Russo and Sabelfeld [34] use small step semantics for their security definitions.

To leverage the results of [34], we need a two-way correspondence between computations of the transformed program and computations of the VM-monitored source program. Steps of the latter correspond to multiple steps of the former (like weak bisimulation), and it is a little tricky to express the correspondence and formulate induction hypotheses for the core lemmas. This kind of correspondence resembles atomicity refinements for concurrent programs, but what we need is equivalence, not just one-way refinement. Two-way correspondence is needed in hardware verification in order to connect a concrete model of a CPU to its abstract instruction set architecture. Our equivalence proof is similar to the verification method of Park, Burch, and Dill [7], [32]. Their technique was applied to transition systems that model a finite set of machine instructions [7], whereas our result pertains to the structured operational semantics of an infinite set of commands. Their technique uses a feature of the concrete processor called "stalling": a particular input signal causes the machine to take steps which eventually flush the pipelines by completing execution of all pending instructions, while not initiating the next instruction. The idea is generalized in [32] to describe an abstract step as a distributed transaction (specifically, for a cache coherence protocol). The idea is to connect an arbitrary state of the concrete machine to the state that would be reached by stalling [7] or running to the transaction commit point [32], and that in turn corresponds to a state of the abstract machine. Our source language does not have a feature like stalling. However, we achieve a similar effect by instrumenting programs with labeled skips that serve to mark control points in source code at which monitoring actions have run to completion.

III. Background

In a paper that appears in this volume [34], Russo and Sabelfeld discuss soundness of flow sensitive information flow policy enforcement. Their main result is the impossibility of sound, purely dynamic, flow sensitive monitoring. They also define a flow sensitive information flow monitor, or rather a family of four variations which differ in how they respond to outputs which would be insecure. The monitor is *hybrid*, in their terminology, meaning that it employs static

analysis (for branches not taken). We take their monitor as the basis of our work, showing how to inline a monitor and leverage the security and transparency properties of the VM monitor.

The choice of this particular VM monitor was made for three reasons. First, it is flow sensitive and the language includes output commands. Second, the definitions are elegantly modular, which in particular makes transparency easy to see (which would be especially important for scaling to a realistic language). Third, a rigorous security proof is available.

In this section, we present the programming language, monitor family, and security property. While keeping close to their formulations, we make minor changes which are described in due course. We explain why the changes do not alter the security property on which we rely.

A. Security policy lattice

For practical purposes, we consider an arbitrary finite lattice, $Levs$, of security labels. The security policy is given by a conventional labeling: initial labels on the program variables (which serve as inputs) and fixed labels on the output channels, as detailed later. A multi-level security lattice provides an elegant way to express security policies, notably in web applications [27].

Russo and Sabelfeld's work [34] uses the two-point lattice, which suffices for their purposes. The security property for a general lattice quantifies over observer levels, and distinguishes high from low levels relative to the observer; that's why, for theory, a two-point lattice is enough. For practical purposes, we don't want to run multiple monitors; we want a single one that handles multiple levels. In Theorem 1 we justify our generalization of the VM monitor to multiple levels.

To cater for multiple levels in the information flow tracking code introduced by the inlining transformation, we slightly extend the programming language of [34]. In addition to integer values and arithmetic operations, we include levels and lattice operations. Of course these could be encoded using integers, and in a richer language other encodings might be used. We distinguish these expressions merely for clarity where we use them in the transformations.[1]

B. Language and semantics

The abstract syntax is presented in Figure 1: expressions e include literal values $v \in Vals$ (integers, security levels), arithmetic operations, level join $e \sqcup e$, level comparison $e \sqsubseteq e$. We use identifier l for literal levels, i.e., elements of $Levs$ which is a subset of $Vals$.

[1]Levels may as well be allowed in the source program. Of course levels manipulated by the monitor are separate from those manipulated by the source program, which have nothing to do with the security property. (This is substantiated by Theorem 2.)

$$
\begin{array}{lrcl}
\textit{Expressions} & e & ::= & v \mid x \mid e \oplus e \\
 & v \in \textit{Vals},\ x \in \textit{Vars},\ \oplus \in \{+, \sqcup, \sqsubseteq, \ldots\} & & \\
\textit{Commands} & c & ::= & \textbf{skip} \mid x \textbf{ := } e \mid c\textbf{; } c \\
 & & \mid & \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c \\
 & & \mid & \textbf{output}_l(e) \mid \square \mid \succ \mid \checkmark
\end{array}
$$

Figure 1. Abstract syntax.

$$
\begin{array}{lrcl}
\text{Command events} & \beta & ::= & \alpha \mid t \\
 & \alpha & ::= & \epsilon \mid o_l(e, v) \\
 & \epsilon & ::= & a(x, e) \mid b(e, c) \mid f \mid s \\
\text{Observations} & \omega & ::= & o_l(v)
\end{array}
$$

Figure 2. Events.

Inputs are modeled by the initial values of program variables. There is an explicit command, $\textbf{output}_l(e)$, that immediately outputs the value of e on the "channel" that is considered to be visible to observers at level l and above.

The command $\square$ has no transitions; it serves merely to mark terminal configurations in the semantics.

Transitions generate internal events that are visible only to the VM monitor. These are the elegant way in which Russo and Sabelfeld define their monitor. The syntax of events is displayed in Figure 2. The transition events consist of the "skip" event s, variable assignment event $a(x, e)$, branching $b(x, e)$, join-of-branches event f, the "step" event t for the $\checkmark$ command, and the event $o_l(e, v)$ for output. The categories α and ϵ are distinguished for later reference, as are β and α.

The commands $\succ$ and $\checkmark$ are essentially labeled skips. Command $\succ$ generates the event f that lets the VM monitor synchronize with joint points in control flow. It is inserted into the configuration by the transitions for branching commands. Both $\succ$ and the preceding events appear in the semantics of Russo and Sabelfeld. The remaining one, $\checkmark$, is our addition.

Command $\checkmark$ is not allowed in source code, so the VM monitor does not need to handle the t event that it emits. Both $\checkmark$ and $\succ$ are used by the inliner. Neither is observable, but they are key ingredients of our correctness proofs.

The semantics of the language is defined in Fig. 3 via labeled, small-step transitions over configurations of the form $\langle c, m \rangle$, where $m : Vars \rightharpoonup Vals$ is a memory. In the initial configuration the domain of m would be $vars(c)$, and its domain is invariant under the semantic transitions.

We extend m homomorphically to expressions: for values, $m(v) = v$; for arithmetic, $m(e + e') = m(e) + m(e')$ etc; for level join, $m(e \sqcup e') = m(e) \sqcup m(e')$ where $\sqcup$ is the lattice join; and $m(e \sqsubseteq e')$ is 0 or 1 according to whether $m(e) \sqsubseteq m(e')$ in $Levs$. These would all be total functions if we encoded levels as integers, and the level expressions

$$\textsc{Skip}\quad \langle \textbf{skip},\, m\rangle \xrightarrow{s} \langle \square,\, m\rangle$$

$$\textsc{Assign}\quad \frac{m(e) = v}{\langle x := e,\, m\rangle \xrightarrow{a(x,e)} \langle \square,\, m[x \mapsto v]\rangle}$$

$$\textsc{Seq1}\quad \frac{\langle c_1,\, m\rangle \xrightarrow{\alpha} \langle \square,\, m'\rangle}{\langle c_1; c_2,\, m\rangle \xrightarrow{\alpha} \langle c_2,\, m'\rangle}$$

$$\textsc{Seq2}\quad \frac{\langle c_1,\, m\rangle \xrightarrow{\alpha} \langle c_1',\, m'\rangle \qquad c_1' \neq \square}{\langle c_1; c_2,\, m\rangle \xrightarrow{\alpha} \langle c_1'; c_2,\, m'\rangle}$$

$$\textsc{IfThen}\quad \frac{m(e) \neq 0}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2,\, m\rangle \xrightarrow{b(e,c_2)} \langle c_1; \succ,\, m\rangle}$$

$$\textsc{IfElse}\quad \frac{m(e) = 0}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2,\, m\rangle \xrightarrow{b(e,c_1)} \langle c_2; \succ,\, m\rangle}$$

$$\textsc{WhileLoop}\quad \frac{m(e) \neq 0}{\langle \textbf{while } e \textbf{ do } c,\, m\rangle \xrightarrow{b(e,\textbf{skip})} \langle c; \succ; \textbf{while } e \textbf{ do } c,\, m\rangle}$$

$$\textsc{WhileSkip}\quad \frac{m(e) = 0}{\langle \textbf{while } e \textbf{ do } c,\, m\rangle \xrightarrow{b(e,c)} \langle \succ,\, m\rangle}$$

$$\textsc{Output}\quad \frac{m(e) = v}{\langle \textbf{output}_l(e),\, m\rangle \xrightarrow{o_l(e,v)} \langle \square,\, m\rangle} \qquad \textsc{End}\quad \langle \succ,\, m\rangle \xrightarrow{f} \langle \square,\, m\rangle$$

$$\textsc{Step}\quad \langle \checkmark,\, m\rangle \xrightarrow{t} \langle \square,\, m\rangle$$

Figure 3. Language semantics. We write $\langle c,\, m\rangle \xRightarrow{\vec{\beta}} \langle c',\, m'\rangle$ for the transitive closure of the transition relation, with $\vec{\beta}$ the catenation of labels.

introduced by inlining will be type-correct —so we refrain from fussing about expression types.

Figure 2 defines the output events, $o_l(v)$, projected from the internal $o_l(e, v)$, that provide the observable behavior of a program. This is ultimately what is used to define the security property. For simplicity, we use a single semantics both for VM monitoring —for which the internal events are needed— and for programs with inlined monitor —for which the internal events are ignored.

$$\textsc{M-skip}\quad \langle \Gamma,\, \Delta\rangle \xrightarrow{s} \langle \Gamma,\, \Delta\rangle$$

$$\textsc{M-assign}\quad \langle \Gamma,\, \Delta\rangle \xrightarrow{a(x,e)} \langle \Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)],\, \Delta\rangle$$

$$\textsc{M-branch}\quad \frac{l = \Gamma(e) \sqcup lev(\Delta)}{\langle \Gamma,\, \Delta\rangle \xrightarrow{b(e,c)} \langle \Gamma,\, (update_l(c) :: \Delta)\rangle}$$

$$\textsc{M-join}\quad \langle \Gamma,\, ((xs, l) :: \Delta)\rangle \xrightarrow{f} \langle \Gamma \sqcup mkFun((xs, l)),\, \Delta\rangle$$

$$\textsc{M-OutputFailstop}\quad \frac{lev(\Delta) \sqcup \Gamma(e) \sqsubseteq l}{\langle \Gamma,\, \Delta\rangle \xrightarrow[o_l(v)]{o_l(e,v)} \langle \Gamma,\, \Delta\rangle}$$

$$\textsc{M-OutputDefault}\quad \frac{\Gamma(e) \sqsubseteq l \implies v' = v \qquad \Gamma(e) \not\sqsubseteq l \implies v' = D}{\langle \Gamma,\, \Delta\rangle \xrightarrow[o_l(v')]{o_l(e,v)} \langle \Gamma,\, \Delta\rangle}$$

$$\textsc{M-OutputSuppress}\quad \frac{lev(\Delta) \sqcup \Gamma(e) \sqsubseteq l \implies \gamma = o_l(v) \qquad lev(\Delta) \sqcup \Gamma(e) \not\sqsubseteq l \implies \gamma \text{ is nothing}}{\langle \Gamma,\, \Delta\rangle \xrightarrow{o_l(e,v)}_{\gamma} \langle \Gamma,\, \Delta\rangle}$$

$$\textsc{M-OutputDefault/Suppress}\quad \frac{lev(\Delta) \not\sqsubseteq l \implies \gamma \text{ is nothing} \qquad lev(\Delta) \sqcup \Gamma(e) \sqsubseteq l \implies \gamma = o_l(v) \qquad lev(\Delta) \sqsubseteq l \wedge \Gamma(e) \not\sqsubseteq l \implies \gamma = o_l(D)}{\langle \Gamma,\, \Delta\rangle \xrightarrow{o_l(e,v)}_{\gamma} \langle \Gamma,\, \Delta\rangle}$$

Figure 4. VM monitor transitions.

C. VM monitor

Our information flow monitor is hybrid and flow sensitive. Hybrid means that it employs static analysis of the conditional branches that were not taken in the current run. Flow sensitive means that security levels of variables could be updated during the run. It is a generalization of a monitor by Russo and Sabelfeld [34] to a multi-level security lattice.

Monitor transitions look like $\langle \Gamma,\, \Delta\rangle \xrightarrow{\alpha}_{\gamma} \langle \Gamma',\, \Delta'\rangle$ and will be explained in due course. The monitor runs alongside the program in a lockstep fashion.

Definition 1 (monitored transitions) The transition relation $\xrightarrow{\gamma}$ on monitored configurations is defined by

$$\frac{\langle c,\, m\rangle \xrightarrow{\alpha} \langle c',\, m'\rangle \qquad \langle \Gamma,\, \Delta\rangle \xrightarrow{\alpha}_{\gamma} \langle \Gamma',\, \Delta'\rangle}{\langle\langle c,\, m\rangle, \langle \Gamma,\, \Delta\rangle\rangle \xrightarrow{\gamma} \langle\langle c',\, m'\rangle, \langle \Gamma',\, \Delta'\rangle\rangle}$$

At each step of the program the monitor inspects the event α generated by the program, updates its state, and generates γ

$$
\begin{array}{lcl}
\mathit{modif}(\textbf{skip}) & = & \emptyset \\
\mathit{modif}(\succ) & = & \emptyset \\
\mathit{modif}(\Box) & = & \emptyset \\
\mathit{modif}(c_1\textbf{;}\ c_2) & = & \mathit{modif}(c_1) \cup \mathit{modif}(c_2) \\
\mathit{modif}(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2) & = & \mathit{modif}(c_1) \cup \mathit{modif}(c_2) \\
\mathit{modif}(\textbf{while}\ e\ \textbf{do}\ c) & = & \mathit{modif}(c) \\
\mathit{modif}(\textbf{output}_l(e)) & = & \emptyset \\
\mathit{modif}(x\ \textbf{:=}\ e) & = & \{x\}
\end{array}
$$

Figure 5. The modifiable variables of a command.

which is either an observable output event (category ω in Figure 2) or nothing. Note that event category α is used in Definition 1 because the VM monitor is defined for source programs that do not include the event $\checkmark$.

For monitored transitions, we define

$$\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \stackrel{\omega}{\Longrightarrow} \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$$
$$\text{iff}$$
$$\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \longrightarrow^* \stackrel{\omega}{\longrightarrow} \longrightarrow^* \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$$

and write $\stackrel{\vec{\omega}}{\Longrightarrow}$ for the transitive closure with catenation of labels.

The transitions of our VM monitor are presented in Figure 4. These labeled transitions relate monitor configurations of the form $\langle\Gamma,\, \Delta\rangle$. Here Γ : $\mathit{Vars} \rightharpoonup \mathit{Levs}$ is a partial function from variable names to security levels, used to track the level of each variable's current value. The *control context* Δ is a non-empty list of pairs of the form (xs, l) where xs is a set of variables and l is a security level; a pair corresponds to a branch point with l being the level of the branch condition and xs being the variables that may be written in the branch that wasn't taken in the current run. (Compare Δ with Γ_s in [34].)

We extend labelings Γ homomorphically to expressions: $\Gamma(e)$ is the join of the levels $\Gamma(x)$ of variables x in e.

We are only concerned with monitored executions which begin with $\langle\Gamma_0,\, \Delta_0\rangle$, where Γ_0 is the initial labeling, Δ_0 is $((\bot, \emptyset) :: [\,])$, and $\bot$ is the bottom level of the security lattice Levs, and where $\mathit{dom}(m) = \mathit{dom}(\Gamma_0)$. We write $[\,]$ for the empty list and use the ML notation :: for prefixing an element to a list.

To create a pair to push onto Δ, we use the function $\mathit{update}_l(c)$ defined by

$$\mathit{update}_l(c) = (\mathit{modif}(c), l) \tag{1}$$

using the set $\mathit{modif}(c)$ of assignment targets of c (defined in Fig. 5). A pair (xs, l) is converted to an l-labeling of the variables xs by mkFun defined by

$$\mathit{mkFun}(xs, l) = \{[x \mapsto l] \mid x \in xs\}$$

Observe the invariant that if $\Delta = ((xs, l) :: \Delta')$ then l is the join of all the levels of the branch conditions that the current command implicitly depends on, i.e., including Δ'. The rule [M-Branch] makes sure of that. The monitor uses a function lev to get the level of the current control dependence region:

$$\mathit{lev}((xs, l) :: \Delta) \quad = \quad l$$

The join $\Gamma \sqcup \Gamma'$ of two labelings has as its domain the union of the domains of Γ and Γ'; it is defined in Fig. 6.

$$
\begin{array}{rll}
(\Gamma \sqcup \Gamma')(x) = \Gamma(x) \sqcup \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma') \\
= \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma) - \mathit{dom}(\Gamma') \\
= \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma') - \mathit{dom}(\Gamma)
\end{array}
$$

Figure 6. Join of partial labelings.

The definitions provide a family of four monitors, all the same except for their treatment of output events. The alternatives are given by the four [M-Output*] rules in Fig. 1. The variations have to do with response to insecure output. The rules also take care of a technical issue: for communication between the monitor and monitored program, we need the internal output event to include the output expression so its levels can be checked. But this should not appear in the actual output, which is captured by events as subscript on the transition arrow (cf. ω in Fig. 2). In [M-OutputDefault], D is some fixed literal value.

Russo and Sabelfeld [34] do not define of the update function used in their monitors; instead that they specify the property required of update. This caters for more sophisticated features such as aliased objects, for which a practical definition would be imprecise. We give a specific definition, disentangled to emphasize that it is modif which would be imprecise in a more sophisticated language. In fact, our soundness proofs only rely on the fact that modif covers possible state changes in the usual sense: for any $m, c, m', c', \vec{\alpha}, x$, if $\langle c,\, m\rangle \stackrel{\vec{\alpha}}{\Longrightarrow} \langle c',\, m'\rangle$ and $x \notin \mathit{modif}(c)$ then $m(x) = m(x')$. One can easily see that by mandating this property of modif we make our definition of update satisfy the requirements in [34].

The main difference between our monitor and that of Russo and Sabelfeld is the fact that our monitor works with a multi-level security lattice. This difference is manifested in the structure of Δ, the definition of lev and in the definition of $\mathit{update}_l(c)$, which is a partial function from variables to security levels, rather than a pair. Theorem 1 says that the differences are not significant with respect to soundness.

D. Security property

Security policy is comprised of the lattice $(\mathit{Levs}, \sqsubseteq)$, the initial labeling Γ_0 of the program variables, and the fixed assignment of levels to output channels. For convenience in the formalization, the labeling of channels is embodied in the program syntax.

Our attacker model is the same as in [34]. We assume that the attacker knows or supplies the monitored programs and has an arbitrary power of reasoning about program semantics. However, the attacker does not have control over the virtual machine, the monitor or the inliner. She cannot observe the values stored in memory directly, nor can she influence the behavior of the system in any way other than supplying the programs that are subject to monitoring/transformation and providing non-secret inputs, which are modeled as the initial values of low variables.

We assume that the attacker has a specific level and can observe all outputs on channels at that level and below. However she is insensitive to power consumption, time or any other covert channels. We believe that the described attacker model fits the threat space of mobile code well. For example, in a typical cross-site scripting scenario the attacker can supply malicious JavaScript code and observe the requests made to a web-site she controls, but she cannot influence the work of the user system directly, nor intercept the communication between the user and the trusted web-site.

The desired security property is *termination insensitive non-interference* (TINI) [34], [2]. It is concerned with what may be learned about the initial values of variables that are initially labeled secret, by an observer who sees low outputs and the initial values of low variables, but nothing else. (In particular, not the internal events, category β in Figure 2.) Termination insensitivity is attractive because it admits more programs; it is adequate because, even in the presence of output, ignoring termination does not allow leakage of secrets in polynomial time [2]. The term "progress insensitivity" is sometimes used, because the formal definition says that what is learned from observing a specific run is no more than what is learned by knowing that there exists a run with the same number of output events.

Definition 2 (TINI) A command c satisfies TINI if the following holds for attackers at any level l. For any initial memories m_1, m_2 that agree on variables x such that $\Gamma_0(x) \sqsubseteq l$, if $\langle c,\, m_1\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle c',\, m_1'\rangle$ then there are m_2' and $\vec{\omega}'$ such that $\langle c,\, m_2\rangle \stackrel{\vec{\omega}'}{\Longrightarrow} \langle c',\, m_2'\rangle$ and either (a) $\vec{\omega}$ and $\vec{\omega}'$ have the same sequences of l-visible events (i.e. outputs on channels of level $\sqsubseteq l$), or (b) the l-visible events of $\vec{\omega}'$ are a prefix of those of $\vec{\omega}$ and configuration $\langle c',\, m_2'\rangle$ is stuck due to the monitor.

Russo and Sabelfeld prove that their monitor is secure: every command satisfies TINI when executed under the monitor. Note that for monitored executions, the definition of TINI is adapted to monitored transitions (Def. 1), using $\langle \Gamma_0,\, \Delta_0\rangle$ as initial monitor configuration.

In section V we establish soundness of our in-lining process by proving that for any source program its transformation produces the same sequence of outputs and terminates at the same time as the monitored program starting in equivalent states. Combined with the result of Russo and Sabelfeld, this implies the security of programs with the monitor inlined.

IV. Inlining

The in-lining process is governed by transformation judgments of the form $G, S, I \vdash c \blacktriangleright \tilde{c}$ for commands. The command rules are displayed in Fig. 8. The inlined version, for which we use identifier $\tilde{c}$, is determined by c together with G, S, and I. As explained below, G, S, and I are static information that describes the layout of a part of program memory which is used to store the monitor state.

Function $S : vars(c) \rightarrow Vars$ maps each variable of c to its *shadow variable* $S(x)$ which will hold the current security level of x. We require that S is injective and its range should be disjoint from $vars(c)$. The finite list G contains names of the variables that store security levels of the conditional guard expressions in the current control dependence region with the $head(G)$ being the security level of the innermost conditional or loop guard. The finite list I holds sets of variables of c that could possibly be updated in the branches-not-taken. We require $|I| = |G|$ and moreover the elements of G are disjoint from $rng(S)$. The transformation rules maintain these conditions.

The inlining rules for commands refer to a judgment of the form $S \vdash e \blacktriangleright \tilde{e}$, for expressions, given by rules in Fig. 9. It's meaning is that $\tilde{e}$ is an expression for the level join of the shadow variables of the variables in e.

In the inlined program, the memory can be partitioned as a memory m for the original variables and a memory mst for the variables in G and $rng(S)$. The following definition describes a correspondence between mst and a VM-monitor configuration. It uses the standard function that converts a pair of lists into a list of pairs:

$$\begin{array}{lcl} zip((a :: as), (b :: bs)) & = & ((a, b) :: zip(as, bs)) \\ zip(as, bs) & = & [\,] \quad \text{if } as = [\,] \text{ or } bs = [\,] \end{array}$$

Definition 3 (monitor state coupling) Define

$$\langle \Gamma,\, \Delta\rangle \cong (G, S, I, mst)$$

iff $\Gamma = mst \circ S$ and $\Delta = zip(I, mst \circ G)$.

(We write $mst \circ G$ for the list obtained by mapping mst over list G; think of G as a function of its indices.)

The transformed program should start in a configuration $\langle \tilde{c},\, m_0 \uplus mst_0\rangle$ where m_0 is the initial user memory and mst_0 is the initial *monitor state*. The monitor state satisfies the invariant: $dom(mst) = rng(S) \cup elements(G)$. The initial monitor state mst_0 should be coupled with $\langle \Gamma_0,\, \Delta_0\rangle$.

We prove that if the initial memory and monitor state are disjoint they stay disjoint along the whole execution of the program.

For a source program c, the transformed version $\tilde{c}$ is obtained by the transformation

$$(x :: [\,]), \mathit{init}(c), (\emptyset :: [\,]) \vdash c \blacktriangleright \tilde{c}$$

where fresh variable x is for the level of the initial security context, $\mathit{init}(c)$ gives us a total injective mapping from $\mathit{vars}(c)$ to a set of variables disjoint from $\mathit{vars}(c)$, and $\emptyset$ is the branch-not-taken set for the initial context.

The transformed programs include commands $\succ$ and ✓ that have no observable behavior and would be omitted by an implementation; for us they serve as annotations to facilitate the correctness proof.

The transformation process introduces a number of additional commands that track explicit and implicit information flows during the execution of the program. These commands mimic the operations that a VM-monitor does at each step of the computation.

For example, the [TR-Assign] rule introduces an extra assignment command that updates a shadow variable $S(x)$ to the least upper bound of the level of the expression e (the value of the transformed expression $\tilde{e}$) and the level of the control context stored in the variable $\mathit{head}(G)$ on top of the stack of guard levels G. The definition of monitor state coupling tells us that $S(x)$ corresponds to $\Gamma(x)$ and the VM monitor updates it to the same security level as the in-lined monitor. The rule also includes a ✓ command which reflects that, after the transformed program updates x's shadow and x itself, the state corresponds to the state of the VM-monitored program after one step.

According to the rule for conditional [TR-If] the program rewriter would first find a fresh variable name x. This variable would store the security level of the conditional guard; the command $x := \tilde{e} \sqcup \mathit{head}(G)$ will make sure of that. Next, it should transform each branch c_i, but in a different environment: x should be pushed on top of the guard level stack G to reflect the control dependency on the guard. Also the branch-not-taken effect stack I should be updated with the set of variables that might be assigned in the other branch in order to account for implicit flows. This mimics the operations that are done by the VM-monitor on the $b(e, c)$ event. A final touch is putting the ✓ commands in the proper places. It is, perhaps, strange to see not one but three ✓ commands introduced by the rule, knowing that each roughly corresponds to one step taken by the VM-monitored program. However, observe that two of these commands are in the branches. Which means only one gets to be executed because only one of the branches could be taken. Among the two ✓ commands introduced by this rule the first one serves to mark the end of evaluating the guard expression and taking the branching decision. The trailing ✓ has to do with the $\succ$ command that the conditional is reduced to when the branch is finished executing. Since it was not present in the original code, the transformation could not be applied to it. However, a $\succ$ command in the VM-monitored execution will take a step which should be matched by a t in the trace of transformed program. Thus, the [TR-If] rule introduces an extra ✓ for the [TR-End] rule. Also, in order to make the [TR-end] rule work, the rule instructs to introduce the commands given by $\kappa(\mathit{modif}(c_{i \bmod 2+1}), S, x)$.

$$\text{TR-VALUE}\quad \frac{}{S \vdash v \blacktriangleright \bot} \qquad \text{TR-VAR}\quad \frac{}{S \vdash x \blacktriangleright S(x)}$$

$$\text{TR-OP}\quad \frac{\oplus \in \{+, \sqcup, \sqsubseteq, \ldots\} \qquad S \vdash e_1 \blacktriangleright \tilde{e}_1 \qquad S \vdash e_2 \blacktriangleright \tilde{e}_2}{S \vdash e_1 \oplus e_2 \blacktriangleright \tilde{e}_1 \sqcup \tilde{e}_2}$$

Figure 9. Transformation rules for expressions.

The functions $\mathit{modif}(c)$ (Figure 5) and $\kappa(i, S, e_L)$ (Figure 7) are used to generate code that is an in-lined implementation of the $\mathit{update}_l(c)$ function. The result of κ performs compensation of the security levels of variables for the implicit flows resulting from the fact that the other branch was not taken.

Now that we have used [TR-Assign] and [TR-If] to illustrate the key principles and components of the transformation rules, we will only highlight the principal ideas behind the rest.

The rule for **while** is similar to the previous rule. We introduce a new variable x to hold the level of the guard which gets updated at the end of each iteration to capture flow sensitivity. The loop body is transformed taking the control dependency on the guard value into account. At the end of the loop there is a compensation $\kappa(\mathit{modif}(c), S, x)$ for the fact that the loop body is not taken. The extra **skip**, $\succ$ and ✓ commands are used in the proof of Theorem 2. All these extra skips could be removed from transformation rules for "production" use since they do not modify the memory, alter the control flow or produce any observable outputs.

The family of rules [TR-Output*] is the direct counterpart of the family of monitor behaviors [M-Output*]. Note that only one of the rules should be used in the actual transformation. As in [34] we leave the user a choice of enforcement strategy while maintaining the same guarantees about the resulting program behavior.

The rules in Figure 10 should not be used in the inlining process. Indeed, they are not applicable to source programs, except for [TR-WhileLoop]. These rules are used in the proof of Theorem 2; they help build correspondence between the commands in the VM configurations and in the inlined monitor configurations during the execution (see Def. 5 below). Their use is best explained by execution traces presented in the proof for Theorem 2.

V. Security and Transparency

We show both security and transparency by proving that transformed programs are *observationally equivalent*

$$\kappa(xs, S, y) = \begin{cases} S(x) := y \sqcup S(x); \kappa(xs \setminus x, S, y) & \text{if } x \in xs \\ \textbf{skip} & \text{if } xs = \emptyset \end{cases}$$

Figure 7. Definition of the compensation function, κ, by recursion on the variable set xs

TR-SKIP
$$G, S, I \vdash \textbf{skip} \blacktriangleright \textbf{skip}; \checkmark$$

TR-ASSIGN
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash x := e \blacktriangleright S(x) := \tilde{e} \sqcup head(G); x := e; \checkmark}$$

TR-SEQ
$$\frac{\forall i \in 1..2 \mid G, S, I \vdash c_i \blacktriangleright \tilde{c_i} \qquad c_1 \neq \succ}{G, S, I \vdash c_1; c_2 \blacktriangleright \tilde{c_1}; \tilde{c_2}}$$

TR-IF
$$\frac{S \vdash e \blacktriangleright \tilde{e} \qquad x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G \qquad \forall i \in 1..2.\ (x :: G), S, (modif(c_{i \bmod 2+1}) :: I) \vdash c_i \blacktriangleright \tilde{c_i} \qquad \forall i \in 1..2.\ \tilde{c_i}' = \checkmark; \tilde{c_i}; \kappa(modif(c_{i \bmod 2+1}), S, x)}{G, S, I \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \blacktriangleright x := \tilde{e} \sqcup head(G); \textbf{if } e \textbf{ then } \tilde{c_1}' \textbf{ else } \tilde{c_2}'; \checkmark}$$

TR-WHILE
$$\frac{S \vdash e \blacktriangleright \tilde{e} \qquad x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G \qquad (x :: G), S, (\emptyset :: I) \vdash c \blacktriangleright \tilde{c}}{G, S, I \vdash \textbf{while } e \textbf{ do } c \blacktriangleright x := \tilde{e} \sqcup head(G); \textbf{while } e \textbf{ do } (\checkmark; \tilde{c}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(modif(c), S, x); \succ; \checkmark}$$

TR-OUTPUTFAILSTOP
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge}); \checkmark}$$

TR-OUTPUTDEFAULT
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark}$$

TR-OUTPUTSUPPRESS
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip}; \checkmark}$$

TR-OUTPUTDEFAULT/SUPPRESS
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip}; \checkmark}$$

Figure 8. Transformation rules for commands

to programs monitored by the VM monitor. Two parallel runs of the monitored and the transformed programs, starting with the same user memory and compatible monitor states, produce the same output traces and end up with the same user memory and compatible monitor states. Transparency and security for inlined programs then follows from the transparency and security of the VM monitor. Transparency is addressed in Sect. V-A and security in Sect. V-B.

A. Transparency

Informally, transparency means that the observable behavior of a monitored program is the same as un-monitored, except that the monitor may halt it prematurely. An attractive feature of the VM monitor as formalized in Sect. III-C is that transparency is obvious from the way monitored transitions are based directly on transitions of the underlying program (Definition 1).

The transition semantics in Fig. 3 includes internal events for synchronization with the monitor. To streamline the paper, we use the same semantics for transformed programs, but for these programs we are only concerned with outputs. Moreover, the internal event $o_l(e, v)$ that represents an output contains an expression e which is not supposed to be observable. Definition 1 serves both to synchronize the monitor with the underlying program and also to convert

TR-STOP
$G, S, I \vdash \square \blacktriangleright \square$

TR-END
$G, S, I \vdash \succ \blacktriangleright \kappa(head(I), S, head(G)); \succ; \checkmark$

TR-ENDSEQ
$$\frac{tail(G), S, tail(I) \vdash c \blacktriangleright \tilde{c}}{G, S, I \vdash \succ; c \blacktriangleright \kappa(head(I), S, head(G)); \succ; \checkmark; \tilde{c}}$$

TR-WHILELOOP
$$\frac{S \vdash e \blacktriangleright \tilde{e} \qquad (x :: G), S, (\emptyset :: I) \vdash c \blacktriangleright \tilde{c}}{G, S, I \vdash \textbf{while } e \textbf{ do } c \blacktriangleright x := \tilde{e} \sqcup x; \succ; \textbf{while } e \textbf{ do } (\checkmark; \tilde{c}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(modif(c), S, x); \succ; \checkmark}$$

Figure 10. Additional transformation rules used in proofs

internal events to observable ones. So, to formalize transparency in terms of the semantics of Fig. 3 we need the following relation which connects traces of internal events with traces of monitored executions.

Definition 4 (Output relation) We define $\vec{\beta} \rhd \vec{\omega}$ to hold iff $\vec{\omega}$ is obtained from $\vec{\beta}$ by deleting non-output events and changing each $o_l(e, v)$ to $o_l(v)$.

Events of category β are included here to cater for the formulation of Theorem 2.

Now we can be a bit more precise about transparency, for the monitor using rule [M-OutputFailstop]. Transparency has two conditions. First, for any trace $\vec{\omega}$ of a monitored execution from a given initial state, there is an un-monitored execution with trace $\vec{\beta}$ such that $\vec{\beta} \rhd \vec{\omega}$. Second, for every un-monitored execution with trace $\vec{\beta}$ and output $\vec{\omega}$ (i.e., $\vec{\beta} \rhd \vec{\omega}$), there is a monitored execution with trace $\vec{\omega}'$ that is a prefix of $\vec{\omega}$; it is a proper prefix in case the monitor stops the run due to security violation.

We refrain from formalizing and proving the transparency result, as it is straightforward.

For the monitors that respond to violations not by stopping but by outputting defaults etc., one can think of variations like transparency up to the first violation.

Transparency is not explicitly formalized in [34]. However, they prove a related property called *permissiveness*: If a program is deemed secure by the flow sensitive static analysis of Hunt and Sands [23], then each un-monitored execution is matched by an observably equivalent monitored execution. The point is that the monitor does not abort or alter runs of such programs.

B. Security

First, we justify in Theorem 1 that our VM monitor is secure for TINI (Definition 2), based on the security theorem of Russo and Sabelfeld [34]. Then we prove Theorem 2 which says the inlined monitor is observationally equivalent to the VM monitor, which implies that inlined programs are secure.

Theorem 1 Our multi-level VM monitor is secure with respect to TINI.

Proof: Here we only sketch the proof. We are going to rely on the fact that the monitor specified by Russo and Sabelfeld is secure. The principal difference between the two monitors is the fact their monitor works only with two security levels, whereas our security level lattice is arbitrary. We can use the monitor in [34] to construct another monitor that would now be observationally equivalent to our monitor. We do this by decomposing the TINI condition in the following way. First, let's fix an attacker that can observe outputs on or below some level l_a. Then, we can partition the set of security levels *Levs* into two disjoint sub-sets, $\mathcal{H}$ and $\mathcal{L}$, where $\mathcal{L} = \{l \mid l \sqsubseteq l_a\}$ and $\mathcal{H} = \{l \mid l \not\sqsubseteq l_a\}$. Now, if a program c and an initial mapping Γ_0 of variables to security levels are rewritten in such a way any level l occurring in c or Γ_0, l is substituted by L if $l \in \mathcal{L}$, and by H if $l \in \mathcal{H}$, where L and H are elements of a two-level lattice such that L $\sqsubseteq$ H, following [34]. Now, we can see that an information-flow monitor presented by Russo and Sabelfeld can enforce non-interference with respect to the chosen attacker.

To quantify over all attacker levels, we construct a product of two-level monitors (using [M-FailStop]), one for each l_a, with configurations notated as $\langle \Gamma_{l_a}, \Delta_{l_a} \rangle$. The level of a variable x is encoded by the collective values of the $\Gamma_{l_a}(x)$. Transitions of the product monitor require each component to synchronize on ϵ events, just like in Def. 1. For an output event $o_l(e, v)$, the response is dictated by the monitor at level l. Call this the *product monitor*.

To justify our VM monitor with respect to the product monitor, we first observe that in a configuration of the product monitor, the collective labels of a particular variable determine its level in the general lattice. Then we check that the transitions of our VM monitor manipulate levels in a corresponding way, and thwart an output just when some of the components of the product monitor would do the same. Thus the product monitor is strongly bisimilar to our multi-level monitor. ■

Theorem 1 and its proof can be seen in two perspectives. On one hand, it justifies that for theoretical purposes it is enough to use a 2-point lattice. On the other hand, for practical purposes we want a single monitor that uses a multi-level lattice and we show that it can be justified using a two-level monitor.

The rest of this section is devoted to the main result, that VM monitoring is equivalent to the inlined program. Theorem 2 is stated in terms of VM and IM configuration coupling, which uses monitor state coupling defined in Sect. IV.

Definition 5 (Configuration coupling)
$\langle\langle c,\, m_1\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \;\sim\; \langle \tilde{c},\, m_2 \uplus mst\rangle$ if and only if $m_1 = m_2$ and there are G, S, I such that $G, S, I \vdash c \blacktriangleright \tilde{c}$ and $\langle\Gamma,\, \Delta\rangle \cong (G, S, I, mst)$

The most difficult result of the paper is that a program with inlined monitor is observationally equivalent to the same program under the VM monitor. The formal statement is a bit complicated, as the formalization involves several kinds of events, not all of which are considered observable.

The condition used in Theorem 2 resembles a familiar notion of bisimulation. The difference is in the fact that the executions do not match each other's steps one for one. In fact, it is impossible for a transformation of a non-trivial program to exactly match the steps of the VM-monitored program because the transformed program is simply larger due to additional commands that update the monitor state and enforce the policy. For a given output trace, a program with an inlined monitor would necessarily make more steps that its VM-monitored counterpart.

For transitions of command configurations, define $\stackrel{\beta}{\Longrightarrow}$ by

$$\langle c,\, m\rangle \stackrel{\beta}{\Longrightarrow} \langle c',\, m'\rangle$$
$$\text{iff}$$
$$\langle c,\, m\rangle \stackrel{\vec{\epsilon}}{\Longrightarrow}\stackrel{\beta}{\longrightarrow}\stackrel{\vec{\epsilon}}{\Longrightarrow} \langle c',\, m'\rangle$$

We write $\stackrel{\vec{\beta}}{\Longrightarrow}$ for the transitive closure, catenating labels.

Theorem 2 Consider any c, $\tilde{c}$, m, m', mst, Γ, and Δ. Suppose $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \sim \langle \tilde{c},\, m \uplus mst\rangle$. Then we have

(a) For all c', Γ', Δ', $\vec{\omega}$, if

$$\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$$

then there is $\vec{\beta}$ such that $\vec{\beta} \rhd \vec{\omega}$ and

$$\langle \tilde{c},\, m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow} \langle \tilde{c}',\, m' \uplus mst'\rangle$$

and

$$\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle \tilde{c}',\, m' \uplus mst'\rangle$$

(b) For all $\tilde{c}'$, mst', $\vec{\beta}$, if

$$\langle \tilde{c},\, m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow}\stackrel{t}{\longrightarrow} \langle \tilde{c}',\, m' \uplus mst'\rangle$$

then there is $\vec{\omega}$ such that $\vec{\beta} \rhd \vec{\omega}$ and

$$\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$$

and

$$\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle \tilde{c}',\, m' \uplus mst'\rangle$$

Proof: We highlight the most interesting cases here. A complete proof appears in the full version of the paper (available online).

Proof of clause **(a)** goes by induction on the number of steps of the VM monitored configuration $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$. For each step, we go by cases on c and compare one step of the monitored configuration to a sequence of steps of the transformed configuration $\langle \tilde{c},\, m \uplus mst\rangle$ finishing with a t-step. In each case we make sure that if a step of the monitored configuration produces an output $o_l(v)$ then there is an output event $o_l(e, v)$ in the trace of the transformed configuration. If no output is produced, both traces should agree on that. To find instances G', S' and I' for which the resulting configurations are coupled, we are typically guided by the need to establish monitor state coupling: $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$. We are also guided by the form of the resulting commands, since we need to connect them according to the transformation $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$.

We sketch the case where c is $x := e$ as an illustration. First we figure out the transformed command. According to the rule [TR-Assign]: $\tilde{c} = S(x) := \tilde{e} \sqcup g; x := e; \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$ and $g = head(G)$. Next, we show a step of the VM-monitored execution and note the event produced and the resulting configuration.

$$\begin{array}{ll} \longrightarrow & \langle\langle x := e,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \\ & \langle\langle\Box,\, m[x \mapsto m(e)]\rangle,\, \langle\Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)],\, \Delta\rangle\rangle \\ = & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta\rangle\rangle \qquad \text{define } c', m', \Gamma' \end{array}$$

Next, we show the trace of the transformed command until the first t-event:

$$\begin{array}{ll} & \langle S(x) := \tilde{e} \sqcup g; x := e; \checkmark,\, m \uplus mst\rangle \\ \stackrel{\alpha_1}{\longrightarrow} & \quad \text{where } \alpha_1 = a(S(x), \tilde{e} \sqcup g) \\ & \langle x := e; \checkmark,\, m \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle \\ \stackrel{\alpha_2}{\longrightarrow} & \quad \text{where } \alpha_2 = a(x, e) \\ & \langle \checkmark,\, m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle \\ \stackrel{t}{\longrightarrow} & \\ & \langle\Box,\, m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle \\ = & \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \qquad \text{define } \tilde{c}', mst' \end{array}$$

By [TR-Stop] (Fig. 10) we obtain a valid transformation relation between c' and $\tilde{c}'$ in this case: $G', S', I' \vdash \Box \blacktriangleright \Box$,

which holds for any G', S' and I'. In this case we can take $G' = G$, $S' = S$, $I' = I$ and get $\langle \Gamma', \Delta \rangle \cong (G, S, I, mst')$ because $\Gamma' = mst' \circ S$ and $\Delta = zip(I, mst' \circ G)$.

In another case, $c =$ **while** e **do** c_1, we need to consider two subcases: $m(e) = 0$ and $m(e) \neq 0$. In each of them we consider two subcases depending on the rule used to find $\tilde{c}$: either [TR-While] or [TR-WhileLoop]. The proof is long, so we only highlight here the trace of $\langle \tilde{c},\ m \uplus mst \rangle$ (Fig. 11) used in the subcase $m(e) \neq 0$ and we assume $G, S, I \vdash c \blacktriangleright \tilde{c}$ according to [TR-While]. We use the same proof tactic as before.

For clause **(b)** of the Theorem, the proof goes by induction on the number of t-events in the trace of the transformed program. For each t-event, we consider the steps

$$\langle \tilde{c},\ m \uplus mst \rangle \stackrel{\alpha}{\Longrightarrow} \stackrel{t}{\longrightarrow} \langle \tilde{c}',\ m' \uplus mst' \rangle$$

leading to it (by unrolling the semantics of $\tilde{c}$). Going by cases on c, and using the assumption

$$\langle\langle c,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle\rangle \sim \langle \tilde{c},\ m \uplus mst \rangle$$

the semantics gives us c', Γ' and Δ' for the step

$$\langle\langle c,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle\rangle \xrightarrow{\alpha'} \langle\langle c',\ m' \rangle,\ \langle \Gamma',\ \Delta' \rangle\rangle$$

and it remains to check that $\alpha' = \alpha$ and

$$\langle\langle c',\ m' \rangle,\ \langle \Gamma',\ \Delta' \rangle\rangle \sim \langle \tilde{c}',\ m' \uplus mst' \rangle$$

Strictly speaking, this is an inner induction on the structure of c, because the transition relation is defined inductively in the case of sequence.

Note that for both clauses **(a)** and **(b)** the proofs take advantage of determinacy of the transition relations. ■

A direct consequence of the two theorems and the soundness results of [34] is the following.

Corollary 1 Every run of a transformed program satisfies termination-insensitive non-interference (Definition 2) with respect to the given policy.

VI. Discussion

Our long term goal is provably secure information flow control for JavaScript. For reasons discussed in the introduction, it needs to be done by an inlined monitor. Inlining was already suggested by Le Guernic et al. [20] but we have not found prior results on provably correct inlining for information flow monitors. As a first step, we investigated a very simple programming language and found that already it is tricky to carry out a detailed soundness argument. We prove security and transparency of the inlined monitor by connecting it with a VM monitor. It seems plausible that a direct proof of security would be more difficult and it seems clear that a direct proof of transparency would be more difficult, because transparency is obvious for the VM monitor.

For a real world language like JavaScript, merely defining the semantics is a major undertaking [25], [26]. For purposes of specification, it is desirable to use a VM monitor which provides modularity and relative ease of proving transparency. Our hope is that the technical devices we have used in our inlining and its correctness proof will facilitate development of provably secure inlined monitors for richer languages including JavaScript.

One benefit from inlining the monitor comes from the ability to optimize the monitoring code, thus reducing the overhead. This was the original motivation for inlining [14]. The techniques for optimization of monitors for access-control involve trying to prove policy adherence statically for parts of the program, thus, eliminating the need to insert run-time checks. For information flow monitoring, the code that tracks level produces the most overhead. It seems plausible to optimize it by constant folding. Then unnecessary checks could be removed by dead code elimination.

We have chosen this particular language and monitor [34] for three reasons: it is flow-sensitive, modular, and provably sound. We believe that flow sensitivity is important for practical applications. The key reason is the demand for compatibility with legacy software; we don't require secure software to be rewritten in a new language or augmented with security annotations which are required by flow-insensitive approaches [23]. However, as discussed in [34], these benefits come with strings attached: the requirement of static analysis of the branches-not-taken. This static analysis is represented by the $modif$ function (Fig. 5). Although the definition of this function is trivial for the given language, it is quite hard to give adequate definitions for practical dynamic languages, as discussed below. Note that these problems, trade-offs, and potential solutions are applicable to both VM and inlined monitors.

The modification of dynamically allocated objects, as in JavaScript, can be handled in the following way. For each field we create a "shadow" field that stores its security level. These fields are maintained in the same way as the "shadow" variables described in this work. The fact that fields and their "shadows" are stored in the same object means that updates via one access path are visible via aliases.[2] The problem with precise and flow-sensitive treatment of objects lies in the fact that it is generally impossible to get a precise set of all the field locations modified by the program in the presence of aliasing. There is a tradeoff between precision and performance in approximating this information. It is a big question to what degree can aliasing information be approximated without causing too many false positives in existing software.

[2]Special caution should be taken if fields could be added at run-time and if the user can enumerate all the fields of an object. Possible solutions include renaming the "shadow" fields if a field with the same name is added and rewriting the enumeration operations in such a way that they do not access the shadow fields.

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup \mathit{head}(G); \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(\mathit{modif}(c_1), S, x); \succ; \checkmark,\ m \uplus \mathit{mst}\rangle \\
\xrightarrow{\alpha_1} & \quad \text{where } \alpha_1 = a(x, \tilde{e} \sqcup \mathit{head}(G)) \text{ and we use } m(e) \neq 0 \\
 & \langle \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(\mathit{modif}(c_1), S, x); \succ; \checkmark, \\
 & m \uplus \mathit{mst}[x \mapsto \mathit{mst}(\tilde{e}) \sqcup \mathit{mst}(\mathit{head}(G))]\rangle \\
\xrightarrow{\alpha_2} & \quad \text{where } \alpha_2 = b(e, \textbf{skip}) \\
 & \langle \checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x; \succ; \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(\mathit{modif}(c_1), S, x); \succ; \checkmark, \\
 & m \uplus \mathit{mst}[x \mapsto \mathit{mst}(\tilde{e}) \sqcup \mathit{mst}(\mathit{head}(G))]\rangle \\
\xrightarrow{t} & \\
 & \langle \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x; \succ; \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(\mathit{modif}(c_1), S, x); \succ; \checkmark, \\
 & m \uplus \mathit{mst}[x \mapsto \mathit{mst}(\tilde{e}) \sqcup \mathit{mst}(\mathit{head}(G))]\rangle \\
= & \\
 & \langle \tilde{c}',\ m \uplus \mathit{mst}'\rangle \qquad \text{define } \tilde{c}', \mathit{mst}'
\end{array}
$$

Figure 11. Trace of a transformation of **while** e **do** c_1

A related issue is that in some dynamic languages, e.g. JavaScript, functions are first class citizens. That means that they can be passed as values and, what is more important for us, could also be redefined at run-time. A sound static analysis will have to decide which functions could be called from a branch that is not taken and analyze the side-effects of all these functions. This is a non-trivial task, but we hope that we can leverage the existing research in information flow analysis to deal with it (e.g., the technique of Shroff et al. [36]).

Dynamic code evaluation is another tricky but necessary feature, and is an important motivation for dynamic monitoring [3]. At first glance it does not seem a hard task to handle. We can rewrite any occurrence of **eval** in such a way that its argument is first transformed according to our inlining algorithm. (This is done in [28].) We can also insert guards that make sure that the evaluated code does not access the sensitive monitor state. But as with objects, the specification of *modif* is tricky. In general, the variables that might be updated by an **eval** is the set of all the variables in the current scope. Clearly, such judgment will cause a lot of false positives. More precise alternatives would require analysis of the current run-time context, e.g. the set of strings that might be passed to an **eval** as arguments, but this is an expensive analysis to perform.

Austin and Flanagan in [4], [5] take a completely different approach to these problems. They propose to get rid of the requirement to do static analysis by trading some amount of flow-sensitivity. In short, their approach, "no sensitive upgrade", prohibits assignments to low variables in high context. Clearly, this could be implemented by inlining using our approach. (It is done in [28], for a language without output.) In short, we will need to modify the rules for conditionals and loops and exclude the compensation for the branches-not-taken. We can also exclude the list I from the transformation environment. Finally, a change should be made to the assignment rule that would act as a security violation, halting execution if the left hand side has a level which is lower that the current control context. While Austin and Flanagan have proved that this approach is sound, it is not clear to us whether it will produce low false positive rate on the existing applications. It is possible that an approach combining both the static analysis and no-sensitive-upgrade would be the most viable. We are currently investigating these questions through experimentation.

Acknowledgments: Ale Russo and Andrei Sabelfeld kindly shared drafts of their work and discussed it with us. Cormac Flanagan shared unpublished work on "no sensitive upgrade". Natarajan Shankar pointed out the connection with the proof technique of Park, Burch, and Dill.

Chudnov was partially supported by Stevens Innovation and Entrepreneurship Fellowship and by NSF award CNS-0627338. Naumann was partially supported by NSF CNS-0627338, CRI-0708330, and CCF-0915611.

References

[1] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *International Symposium on Formal Methods*, volume 5014 of *Springer LNCS*, pages 262–277, 2008.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *European Symposium on Research in Computer Security*, volume 5383 of *Springer LNCS*, pages 333–348, 2008.

[3] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Symposium*, pages 43–59, 2009.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM Workshop on Programming Languages and Analysis for Security*, pages 113–124, 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, University of California, Santa Cruz, 2009.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symposium on Principles of Programming Languages*, pages 14–25, 2004.

[7] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Conference on Computer Aided Verification*, volume 818 of *Springer LNCS*, pages 68–80, 1994.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM Conference on Programming Language Design and Implementation*, pages 50–62, 2009.

[9] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[10] E. S. Cohen. Information transmission in sequential programs. In *Foundations of Secure Computation*, pages 297–335, 1978.

[11] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded Java. In *European Conference on Object-Oriented Programming*, pages 546–569, 2009.

[12] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[13] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference*, pages 382–391, Dec. 2009.

[14] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95. ACM Press, 1999.

[15] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[16] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[17] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[18] Google. V8 project page. http://code.google.com/p/v8/.

[19] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *IEEE Computer Security Foundations Symposium*, pages 218–232, 2007.

[20] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Advances in Computer Science: Secure Software and Related Issues, 11th Asian Computing Science Conference 2006 (Revised Selected Papers)*, volume 4435 of *Springer LNCS*, pages 75–89, 2008.

[21] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *ACM Workshop on Programming Languages and Analysis for Security*, pages 7–16, 2006.

[22] K. W. Hamlen, J. G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[23] S. Hunt and D. Sands. On flow-sensitive security types. In *ACM Symposium on Principles of Programming Languages*, pages 79–90, 2006.

[24] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *ICISS*, volume 5352 of *Springer LNCS*, pages 56–70, 2008.

[25] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *Springer LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[26] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *IEEE Computer Security Foundations Symposium*, pages 77–91, 2009.

[27] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 15–23, 2010.

[28] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *IFIP International Information Security Conference (SEC)*, Sept. 2010. To appear.

[29] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, 1994.

[30] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, Feb. 2008.

[31] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium*, 2007.

[32] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory Comput. Syst.*, 31(4):355–376, 1998.

[33] J. Richter. *CLR Via C#*. Microsoft Press, 2006.

[34] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *IEEE Computer Security Foundations Symposium*, 2010. This volume.

[35] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[36] P. Shroff, S. F. Smith, and M. Thober. Securing information flow via dynamic capture of dependencies. *J. Comput. Secur.*, 16(5):637–688, 2008. Preliminary version appeared in CSF 2007.

[37] M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Springer LNCS*, pages 312–327, 2010.

[38] Sun Microsystems. The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[39] D. Vanoverberghe and F. Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Springer LNCS*, pages 240–258, 2008.

[40] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Information and Communications Security (ICICS)*, volume 4307 of *Springer LNCS*, pages 332–351, 2006.

[41] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *ACM Symposium on Principles of Programming Languages*, pages 237–249, 2007.

[42] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[43] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *ACM Symposium on Operating Systems Principles*, pages 263–278, 2006.

[44] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy Scope: A precise information flow tracking system for finding application leaks. Technical Report UCB/EECS-2009-145, EECS Department, University of California, Berkeley, Oct 2009.