Static Analysis for Efficient Hybrid Information-Flow Control

Static analysis for efficient hybrid information-flow control

Scott Moore
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA, USA
sdmoore@fas.harvard.edu

Stephen Chong
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA, USA
chong@seas.harvard.edu

***Abstract*—Hybrid information-flow monitors use a combination of static analysis and dynamic mechanisms to provide precise strong information security guarantees. However, unlike purely static mechanisms for information security, hybrid information-flow monitors incur run-time overhead. We show how static analyses can be used to make hybrid information-flow monitors more efficient, in two ways.**

First, a simple static analysis can determine when it is sound for a monitor to stop tracking the security level of certain variables. This potentially reduces run-time overhead of the monitor, particularly in applications where sensitive (i.e., confidential or untrusted) data is infrequently introduced to the system.

Second, we derive sufficient conditions for soundly incorporating a wide range of memory abstractions into information-flow monitors. This allows the selection of a memory abstraction that gives an appropriate tradeoff between efficiency and precision. It also facilitates the development of innovative and sound memory abstractions that use run-time security information maintained by the monitor.

We present and prove our results by extending the information-flow monitor of Russo and Sabelfeld (2010). These results bring us closer to efficient, sound, and precise enforcement of information security.

***Keywords*-information-flow control; hybrid information-flow monitors; dynamic information-flow monitors.**

I. Introduction

Tracking and controlling the flow of information in computer systems can be used to enforce strong, precise, application-specific information security [1]. Information-flow control can be achieved through static or dynamic mechanisms. Static mechanisms (exemplified by *security type systems*, e.g., [2, 3, 4]) analyze a program before execution to determine whether all possible executions are secure. Dynamic mechanisms (e.g., [5, 6, 7]) monitor or instrument program execution to determine whether a particular execution is secure. Dynamic mechanisms can thus be more precise (since they accept or reject a single program execution, instead of an entire program), but unlike static mechanisms, they incur runtime overhead.

Recent work considers *hybrid* information-flow control [8, 9, 10, 11], which combines both static and dynamic mechanisms to enforce information-flow security guarantees. Static mechanisms can reason more precisely than purely dynamic mechanisms about certain kinds of information flows, a result recently proved by Russo and Sabelfeld [11]. Hybrid mechanisms can accept or reject a single program execution, but also reason precisely about information flow.

In this work, we show how static analysis can be used to make hybrid information-flow monitors more efficient in two ways. First, we show that a straight-forward security-type system can reduce runtime overhead of information-flow monitors by determining when it is sound for a monitor to stop tracking certain variables. Second, we derive sufficient conditions for soundly incorporating a wide range of memory abstractions into information-flow monitors. This allows the selection of a memory abstraction that gives an appropriate tradeoff between efficiency and precision. We present and prove these results by extending the fail-stop information-flow monitor of Russo and Sabelfeld [11].

```
1  if (username = "guest") {
2      d := (untrusted)request.data;  // untrusted data
3      untrustedLog := concat(untrustedLog, d)
4  } else {
5      d := request.data;  // trusted data
6  }
7  ...
8  if (action = "update") {
9      updateDB(d); // dangerous operation
10 } else {
11     ...  // no dangerous operations
12 }
13 ...  // no dangerous operations
```

Figure 1. Example of inefficient information-flow monitoring

Selective tracking. Consider the pseudo-code in Figure 1, which models the processing of a web application request. Depending on whether the current user is a guest or an authenticated user, data from the request is regarded as either untrusted or trusted. If the data is untrusted, it is appended to an audit log. After some computation, the data may be used in a database update, which is a dangerous operation that should depend only on trusted data.

Suppose the program executes with a monitor that tracks the flow of information in the program in order to prevent security violations, such as a database update depending on

untrusted data. There are at least three ways in which this information-flow monitor may waste effort at runtime.

First, information in variable untrustedLog never affects whether a security violation occurs, even though it stores untrusted data. Thus, a monitor that does not track variable untrustedLog will still correctly prevent security violations, and so effort spent tracking this variable is wasted. Second, variable d contains untrusted information only on some executions. Depending on the usage of the web application, it may contain trusted data on the majority of executions. In such cases there is no need for the monitor to track variable d. Third, even on executions in which d contains untrusted information, if control reaches line 11 or line 13, then the contents of d can never affect whether a security violation occurs, so there is no need to track it anymore.

There are several opportunities for reducing the number of variables tracked by the monitor, which will, for many monitors, reduce runtime overhead. Even more opportunities exist for a monitor that can dynamically start and stop tracking variables. (We are developing an inlined monitor that dynamically generates instrumented code to track a subset of program variables, and can thus dynamically start and stop tracking variables.)

We present a static analysis that can determine when a program variable is no longer a security concern and show how this analysis can be incorporated into an information-flow monitor. The modified monitor provides exactly the same security guarantee, but with potentially reduced runtime overhead. For the example program above, the analysis enables the monitor to never track variable untrustedLog and to stop tracking variable d when line 11 or 13 is reached. The analysis can either be performed prior to execution or on-the-fly, allowing its use with dynamic languages [12].

```
1   // Create new locations, with empty strings as initial values
2   secretLog := new(" ");
3   normalLog := new(" ");
4   ...
5   if (isSecretAgent(username)) {  // Sensitive information
6       pLog := secretLog;
7   } else {
8       pLog := normalLog;
9   }
10  ...
11  *pLog := concat(*pLog, "logged in");
```

Figure 2. Example of information flow through pointer value

Memory abstractions. Practical information-flow control must deal with realistic language features, including dynamically allocated memory and first-class memory references. Consider the program fragment in Figure 2, which models a web application in which some users are secret agents. If the current user is a secret agent then events are logged in special log to avoid revealing secret activities. Variable pLog points to either secretLog or normalLog and indicates which log will be updated. It is set based on secret information. At line 11, an entry is added to the log. Suppose the current user is a secret agent. Then the location normalLog will not be updated. However, an observer of normalLog may notice that the log is unchanged, and thus learn that the current user is a secret agent. This information flow occurs because the pointer pLog depends on secret information, and updating through this pointer means that learning the value of any location that pLog *might* have pointed to may now reveal secret information.

Monitors can soundly track the flow of information in memory, including the heap, using appropriate *memory abstractions*. The choice of memory abstraction can affect the precision and efficiency of the resulting monitor. In general, fine-grained abstractions may enforce security guarantees precisely, but require an expensive analysis and higher monitor overhead (due to the need to track more abstract locations). Coarser abstractions may be more efficient, but insufficiently precise. In order to explore these tradeoffs, we must first understand the requirements for soundly incorporating a memory abstraction into an information-flow monitor. In addition, we would like to consider to what extent information about the current execution can be used to increase the precision of a memory abstraction.

We present sufficient conditions for incorporating memory abstractions and analyses into a hybrid information-flow monitor. The key insight is that the behavior of the monitor must not leak information, meaning that we must be able to reason statically and precisely about the behavior of the monitor in other possible executions of the program. This has the effect of restricting the precision of memory abstractions. However, the conditions are sufficiently lenient to allow many different memory abstractions to be soundly incorporated into information-flow monitors, such as points-to sets (e.g., [13, 14, 15, 16]), shape graphs (e.g., [17]), and regions (e.g., [18, 19]). The conditions also allow the monitor to use information from the current execution of a program.

The rest of the paper is organized as follows. We present background information in Section II, including Russo and Sabelfeld's fail-stop hybrid information-flow monitor [11]. In Section III we show how an information-flow monitor can soundly stop tracking variables when they are no longer a security concern. In Section IV we show how a wide variety of memory abstractions and analyses can be soundly incorporated into information-flow monitors. We discuss related work in Section V and conclude in Section VI.

II. Background: Information-flow Monitor

In this section, we present a simple imperative language and a hybrid information-flow monitor for the language, based closely on that of Russo and Sabelfeld [11]. We

generalize their language and monitor to an arbitrary security lattice (instead of the two-point lattice they use) and modify the language and monitor slightly to facilitate extensions described in later sections. The modified monitor satisfies the same termination-insensitive noninterference condition as the original monitor, modulo generalization to an arbitrary security lattice. We state the generalized security condition and prove the modified monitor satisfies it.

A. Language

Values	v	::=	n
Expressions	e	::=	$v \mid x \mid e_1 \oplus e_2$
Commands	c	::=	$\mathsf{skip} \mid x := e \mid c_1; c_2 \mid \mathsf{output}_\ell(e) \mid$ $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c$
Terms	t	::=	$c \mid \mathsf{end} \mid \mathsf{stop} \mid t_1; t_2$

Figure 3. Simple imperative language

The syntax for a simple imperative language with an explicit output command is given in Figure 3. Values are restricted to integers n, and expressions e are either values v, variables x, or binary expressions $e_1 \oplus e_2$, where $\oplus$ ranges over total binary operations over integers.

We assume a complete lattice $\mathcal{L}$ of security levels and use $\sqcup$ and $\sqsubseteq$ respectively to denote the join operator and partial order. We write $\top_\mathcal{L}$ and $\bot_\mathcal{L}$ to denote the top and bottom elements of $\mathcal{L}$, respectively. Security levels may represent confidentiality levels, integrity levels, or both. Security levels mandate restrictions on the use of data. Intuitively, levels higher in the lattice mandate more restrictions. For confidentiality, higher in the lattice corresponds to more confidential; for integrity, higher means less trusted. We use the term "sensitive information" to refer to information labeled with a high security level: either confidential information or untrusted information.

Commands c are standard, with the exception of the $\mathsf{output}_\ell(e)$ command, which outputs the value of expression e to channel ℓ, where ℓ is a security level and is intended to be an upper bound on information that may be learned by observing the channel. We assume, without loss of generality, that there is exactly one channel per security level.

Execution of commands introduces new syntactic forms. Terms t represent commands in the process of execution, and extend commands with sequences of terms, stop, and end. Term stop represents a command that has finished execution, and term end indicates that execution is leaving the scope of a branch. The operational semantics inserts end terms when any branch is encountered.

A *program configuration* is a pair $\langle t, m\rangle$, where t is a term and memory m is a mapping from variables to values. The judgment $e, m \Downarrow v$ indicates that expression e evaluates to value v under memory m. We write $m[x \mapsto v]$ for the memory that maps x to v, and otherwise behaves the same as m.

The judgment $\langle t, m\rangle \xrightarrow{\alpha} \langle t', m'\rangle$ indicates that configuration $\langle t, m\rangle$ can take a single step to configuration $\langle t', m'\rangle$. As part of that step, *internal event* α is emitted; the monitor uses internal events to track and control execution. We describe the different internal events in the following subsection. Inference rules for the operational semantics are given in Figure 4.

B. Hybrid information-flow monitor

An information-flow monitor tracks and controls the flow of information in a program. A *monitor configuration* is a pair $\langle \gamma, \sigma\rangle$ consisting of *monitor environment* γ and *monitor stack* σ. Monitor environment γ maps program variables to security levels, tracking the security level of information currently stored in each variable. We write $lev(e, \gamma)$ for the join of levels $\gamma(x)$ for all variables $x \in \mathrm{dom}(\gamma) \cap \mathrm{vars}(e)$, where $\mathrm{vars}(e)$ is the set of all variables that occur in e. Function $lev(e, \gamma)$ gives an upper bound on the information that may be learned by evaluating expression e, assuming that γ describes upper bounds of information stored in variables.

Monitor stack σ is used to track the security level of the program counter, and to account for *implicit flows* [20] (information flows due to the control structure of the program). It is a stack of pairs (ℓ, γ') where ℓ is a security level and γ' is a monitor environment. A pair is pushed on the stack when the program branches and is popped off the stack at the end of the branch. An upper bound on the security level of information influencing control flow can be obtained by taking the join of the first element of all pairs in the stack, denoted $lev(\sigma)$. We refer to this upper bound as the *program counter level*. We describe the use of the monitor stack in more detail below.

A *monitored program configuration* is a pair of a program configuration and a monitor configuration. *Monitored execution* of a program requires that both the program configuration and the monitor configuration can take a step. Figure 5 shows the inference rule for monitored execution. Note that the monitored execution rule is parameterized by monitor M. This allows us to discuss the behaviors of different monitors. We use M_{RS} to denote Russo and Sabelfeld's fail-stop monitor [11].

$$\frac{\langle t, m\rangle \xrightarrow{\alpha} \langle t', m'\rangle \quad \langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M}]{\alpha, t', m}_{\beta} \langle \gamma', \sigma'\rangle}{\langle\langle t, m\rangle, \langle \gamma, \sigma\rangle\rangle \xrightarrow[\mathrm{M}]{\beta} \langle\langle t', m'\rangle, \langle \gamma', \sigma'\rangle\rangle}$$

Figure 5. Semantics of monitored executions

A monitor configuration takes a step based on the internal event generated by the executing program and may produce an output. The small step judgment for monitor configurations is written $\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M}]{\alpha, t', m'}_{\beta} \langle \gamma', \sigma\rangle$, where M indicates which monitor is in use, α is the triggering internal event,

$$\frac{m(x) = v}{x, m \Downarrow v} \qquad \frac{}{v, m \Downarrow v} \qquad \frac{e_1, m \Downarrow v_1 \qquad e_2, m \Downarrow v_2 \qquad v_1 \oplus v_2 = v}{e_1 \oplus e_2, m \Downarrow v} \qquad \frac{}{\langle \mathsf{skip}, m\rangle \xrightarrow{\text{skip}} \langle \mathsf{stop}, m\rangle}$$

$$\frac{e, m \Downarrow v}{\langle x := e, m\rangle \xrightarrow{\text{assign}(x,e)} \langle \mathsf{stop}, m[x \mapsto v]\rangle} \qquad \frac{\langle t_1, m\rangle \xrightarrow{\alpha} \langle \mathsf{stop}, m'\rangle}{\langle t_1; t_2, m\rangle \xrightarrow{\alpha} \langle t_2, m'\rangle} \qquad \frac{\langle t_1, m\rangle \xrightarrow{\alpha} \langle t_1', m'\rangle \qquad t_1' \neq \mathsf{stop}}{\langle t_1; t_2, m\rangle \xrightarrow{\alpha} \langle t_1'; t_2, m'\rangle}$$

$$\frac{e, m \Downarrow v \qquad v \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m\rangle \xrightarrow{\text{branch}(e,c_2)} \langle c_1; \mathsf{end}, m\rangle} \qquad \frac{e, m \Downarrow 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m\rangle \xrightarrow{\text{branch}(e,c_1)} \langle c_2; \mathsf{end}, m\rangle}$$

$$\frac{}{\langle \mathsf{end}, m\rangle \xrightarrow{\text{join}} \langle \mathsf{stop}, m\rangle} \qquad \frac{e, m \Downarrow v \qquad v \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m\rangle \xrightarrow{\text{branch}(e,\mathsf{skip})} \langle c; \mathsf{end}; \mathsf{while}\ e\ \mathsf{do}\ c, m\rangle}$$

$$\frac{e, m \Downarrow 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m\rangle \xrightarrow{\text{branch}(e,c)} \langle \mathsf{end}, m\rangle} \qquad \frac{e, m \Downarrow v}{\langle \mathsf{output}_\ell(e), m\rangle \xrightarrow{\text{output}_\ell(e,v)} \langle \mathsf{stop}, m\rangle}$$

Figure 4. Language semantics

and β is the resulting output, which is either nothing or $\mathrm{o}_\ell(v)$, indicating output of value v on channel ℓ. The monitor can thus halt the execution of the program or modify output in order to enforce security. The small step judgment for monitor configurations also takes the new term t' and the original memory m; this information is not used by Russo and Sabelfeld's monitor but is used by our extensions.

ASSIGN

$$\frac{lev(e,\gamma) \sqcup lev(\sigma) = \ell}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{RS}}]{\text{assign}(x,e),t',m'} \langle \gamma[x \mapsto \ell], \sigma\rangle}$$

BRANCH

$$\frac{lev(e,\gamma) \sqcup lev(\sigma) = \ell}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{RS}}]{\text{branch}(e,c),t',m'} \langle \gamma, (\ell, \textsc{Update}_\ell(c)) : \sigma\rangle}$$

JOIN

$$\frac{}{\langle \gamma, (\ell, \gamma') : \sigma\rangle \xrightarrow[\mathrm{M_{RS}}]{\text{join},t',m'} \langle \gamma \sqcup \gamma', \sigma\rangle}$$

OUTPUT

$$\frac{lev(\sigma) \sqcup lev(e,\gamma) \sqsubseteq \ell}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{RS}}]{\text{output}_\ell(e,v),t',m'}_{\mathrm{o}_\ell(v)} \langle \gamma, \sigma\rangle}$$

SKIP

$$\frac{}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{RS}}]{\text{skip},t',m'} \langle \gamma, \sigma\rangle}$$

Figure 6. Flow-sensitive monitor, $\mathrm{M_{RS}}$

Small step semantics for $\mathrm{M_{RS}}$ are given in Figure 6. Event skip, generated when the program executes a skip command, is always accepted by the monitor and does not change the monitor configuration. Event $\mathrm{assign}(x, e)$ is generated by execution of command $x := e$. The monitor updates the security level of x to the join of the expression's security level $lev(e, \gamma)$ with the program counter level $lev(\sigma)$.

When the program enters a branch (either an if or while command), an internal event $\mathrm{branch}(e, c)$ is generated, where expression e is the branch test, and command c is the branch not taken. This causes the monitor to push a new pair (ℓ, γ') onto the monitor stack, where ℓ is the join of security level of expression e with the current program counter level $lev(\sigma)$, and γ' is the result of calling $\textsc{Update}_\ell(c)$. The function $\textsc{Update}_\ell(c)$ analyses command c and returns a monitor environment γ' such that for every variable x, $\gamma'(x) \in \{\ell, \bot_{\mathcal{L}}\}$, and if c contains an assignment to x then $\gamma'(x) = \ell$, otherwise $\gamma'(x) = \bot_{\mathcal{L}}$.

When the end of a branch is reached, internal event join is generated, which causes the top element (ℓ', γ') of the monitor stack to be popped, and the current monitor environment γ changed to $\gamma \sqcup \gamma'$ (i.e., the point-wise join of γ and γ'). This allows the monitor to track information flows that occur due to code in the branch not taken that could have executed.

Internal event $\mathrm{output}_\ell(e, v)$ is generated when the program attempts to output expression e on channel ℓ. The monitor allows the output only if the information that may be learned by the evaluation of the expression (i.e., $lev(e, \gamma)$) and the information that has influenced the decision to perform the output (i.e., program counter level $lev(\sigma)$) are bounded above by ℓ, the information allowed to be output on the channel. In this paper, we consider only Russo and Sabelfeld's fail-stop monitor; they also explore monitors that behave differently on output events. Our results apply to those other monitor behaviors.

C. Security

Russo and Sabelfeld's fail-stop monitor satisfies a termination-insensitive noninterference security condition. Here we state the noninterference condition, extended in a straightforward way to an arbitrary lattice of security levels, and prove the generalized monitor satisfies the condition.

We write

$$\langle\langle t, m\rangle, \langle\gamma, \sigma\rangle\rangle \xrightarrow[\mathrm{M}]{\vec{\beta}}{}^{*} \langle\langle t', m'\rangle, \langle\gamma', \sigma'\rangle\rangle$$

when monitored program configuration $\langle\langle t, m\rangle, \langle\gamma, \sigma\rangle\rangle$ may take zero or more steps to reach configuration $\langle\langle t', m'\rangle, \langle\gamma', \sigma'\rangle\rangle$, producing the sequence of outputs $\vec{\beta}$. We write $\ell(\vec{\beta})$ for the subsequence of $\vec{\beta}$ containing all and only events output to channels ℓ' such that $\ell' \sqsubseteq \ell$. Intuitively, $\ell(\vec{\beta})$ is the output that is observable during program execution on channels at level ℓ or lower.

Given monitor environment γ, we say that two memories m_1 and m_2 are *ℓ-equivalent*, written $m_1 =^{\ell}_{\gamma} m_2$, if they agree on the values of all variables x such that $\gamma(x) \sqsubseteq \ell$: $\forall x.\ \gamma(x) \sqsubseteq \ell \Rightarrow m_1(x) = m_2(x)$.

Informally, a monitor is secure if for every command c and security level ℓ, given two memories that are ℓ-equivalent, the monitored execution of c from these two memories will produce outputs that look the same to an entity that can observe all channels at level ℓ or lower. Given output sequence $\vec{\beta_1}$, output sequence $\vec{\beta_2}$ looks the same if either $\ell(\vec{\beta_1}) = \ell(\vec{\beta_2})$, or $\ell(\vec{\beta_2})$ is a prefix of $\ell(\vec{\beta_1})$ and no additional observable events will be generated by the execution that produced $\ell(\vec{\beta_2})$. More precisely, we write $\langle\langle t, m\rangle, \langle\gamma, \sigma\rangle\rangle \Rightarrow_{\not\sqsubseteq \ell}$ if for all monitored executions starting from configuration $\langle\langle t, m\rangle, \langle\gamma, \sigma\rangle\rangle$, there are no outputs to a channel at level ℓ or lower.

Definition 1 (Security)**.** *Given monitor* M *and security lattice $\mathcal{L}$,* M *is* secure *if for all commands c, all $\ell \in \mathcal{L}$, all memories m_1 and m_2, and monitor environments γ such that $m_1 =^{\ell}_{\gamma} m_2$, if*

$$\langle\langle c, m_1\rangle, \langle\gamma, \sigma\rangle\rangle \xrightarrow[\mathrm{M}]{\vec{\beta_1}}{}^{*} \langle\langle t_1', m_1'\rangle, \langle\gamma_1', \sigma_1'\rangle\rangle,$$

then there exist t_2', m_2', γ_2', and σ_2' such that

$$\langle\langle c, m_2\rangle, \langle\gamma, \sigma\rangle\rangle \xrightarrow[\mathrm{M}]{\vec{\beta_2}}{}^{*} \langle\langle t_2', m_2'\rangle, \langle\gamma_2', \sigma_2'\rangle\rangle$$

where the following conditions hold.

1) $|\ell(\vec{\beta_2})| \leq |\ell(\vec{\beta_1})|$
2) *if* $|\ell(\vec{\beta_2})| = |\ell(\vec{\beta_1})|$*, then* $\ell(\vec{\beta_1}) = \ell(\vec{\beta_2})$
3) *if* $|\ell(\vec{\beta_2})| < |\ell(\vec{\beta_1})|$*, then* $\ell(\vec{\beta_2})$ *is a prefix of* $\ell(\vec{\beta_1})$ *and* $\langle\langle t_2', m_2'\rangle, \langle\gamma_2', \sigma_2'\rangle\rangle \Rightarrow_{\not\sqsubseteq \ell}$.

Theorem 1. *Monitor* M_{RS} *is secure.*

Due to our extension from a two-point lattice to an arbitrary lattice $\mathcal{L}$, the proof of Theorem 1 given by Russo and Sabelfeld [11] does not apply. However, the theorem holds as an immediate consequence of Theorem 3 in Section IV, which proves security for a language that is a superset of the language described here.

III. Selective tracking

Monitored execution can be significantly slower than normal execution of a program. For example, Chandra and Franz [10] report that with their hybrid information-flow monitor, assignments are $3\times$ slower than unmonitored execution, despite using a restricted set of security levels with an efficient representation. Newsome and Song [21] instrument binaries with information-flow tracking and report CPU-bound computation taking more than $10\times$ longer than executing the binary in the same instrumentation framework with no tracking. Much of the runtime overhead of an information-flow monitor results from tracking the security levels of many variables.

If at some point in the program's execution, the contents of a specific variable can no longer influence the occurrence of a security violation, there is no need to track its security level. By not tracking a variable, the monitor can reduce the overhead associated both with storage for that variable and with performing join operations on its security level. This reduction may be significant for applications where sensitive data is rarely introduced to the system, or where operations that may violate security are rare.

We present a static analysis that soundly determines when a variable can no longer influence the occurrence of a security violation and show that the analysis can be incorporated into an information-flow monitor with no loss of security. We also discuss implementation issues.

A. Static analysis

We use a simple flow-sensitive security type system [22] to determine when a variable cannot cause a security violation. We use a special two-point lattice, containing elements $\bot$ and $\top$, where $\bot \sqsubseteq \top$ and $\top \not\sqsubseteq \bot$. A *typing environment* Γ maps each variable to either $\bot$ or $\top$. Intuitively, if $\Gamma(x) = \top$ then the value stored in variable x may have been influenced by a variable we are considering no longer tracking.

The judgment $\Gamma \vdash e : \tau$ means that under typing environment Γ, information associated with expression e is at most τ. If $\tau = \top$, then the evaluation of the expression may depend on a variable we are considering no longer tracking; if $\tau = \bot$, then evaluation is independent of such variables.

The judgment $pc \vdash \Gamma\ \{t'\}\ \Gamma'$ means that if Γ is an accurate description of the information stored in variables before t' executes, and if $pc \in \{\bot, \top\}$ indicates whether the decision to execute t' depends on variables we are considering no longer tracking, then Γ' will accurately describe the information stored in variables after t' executes.

Inference rules for both judgments are given in Figure 7. We extend $\sqsubseteq$ pointwise for environments and write $\Gamma \sqsubseteq \Gamma'$ to denote $\forall x.\ \Gamma(x) \sqsubseteq \Gamma'(x)$. We write $\Gamma[x \mapsto \tau]$ for

$$\frac{\tau = \bigsqcup_{x \in \mathrm{vars}(e)} \Gamma(x)}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash e : \tau}{pc \vdash \Gamma\,\{x := e\}\,\Gamma[x \mapsto pc \sqcup \tau]}$$

$$\frac{\Gamma \vdash e : \bot}{\bot \vdash \Gamma\,\{\mathsf{output}_\ell(e)\}\,\Gamma} \qquad \frac{pc \vdash \Gamma\,\{t_1\}\,\Gamma' \quad pc \vdash \Gamma'\,\{t_2\}\,\Gamma''}{pc \vdash \Gamma\,\{t_1; t_2\}\,\Gamma''}$$

$$\frac{\Gamma \vdash e : \tau \qquad pc \sqcup \tau \vdash \Gamma\,\{c_i\}\,\Gamma_i \qquad \Gamma_i \sqsubseteq \Gamma' \qquad i = 1, 2}{pc \vdash \Gamma\,\{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\}\,\Gamma'}$$

$$\frac{\Gamma \sqsubseteq \Gamma' \qquad \Gamma' \vdash e : \tau \qquad pc \sqcup \tau \vdash \Gamma'\,\{c\}\,\Gamma'' \qquad \Gamma'' \sqsubseteq \Gamma'}{pc \vdash \Gamma\,\{\mathsf{while}\ e\ \mathsf{do}\ c\}\,\Gamma'}$$

$$\frac{}{pc \vdash \Gamma\,\{\mathsf{skip}\}\,\Gamma} \qquad \frac{}{pc \vdash \Gamma\,\{\mathsf{stop}\}\,\Gamma} \qquad \frac{}{pc \vdash \Gamma\,\{\mathsf{end}\}\,\Gamma}$$

Figure 7. Flow-sensitive information-flow type system

$$\frac{X \subseteq \mathrm{dom}(\gamma') \qquad \langle \gamma, \sigma \rangle \xrightarrow[\mathrm{M_{RS}}]{\alpha, t', m}_{\beta} \langle \gamma', \sigma' \rangle \qquad \bot \vdash \overline{\bot}[X \mapsto \top]\,\{t'\}\,\Gamma}{\langle \gamma, \sigma \rangle \xrightarrow[\mathrm{M_{PERF}}]{\alpha, t', m}_{\beta} \langle \gamma' \setminus X, \sigma' \rangle}$$

Figure 8. Monitor using static analysis to increase performance, $\mathrm{M_{PERF}}$

the typing environment that maps x to τ and otherwise behaves like Γ. More generally, we write $\Gamma[X \mapsto \tau]$ for the environment that maps all variables in the set X to τ and otherwise behaves like Γ. Finally, we write $\overline{\bot}$ for the typing environment that maps every variable to $\bot$.

The inference rules are standard for a flow-sensitive security type system, with the exception of the rule for command $\mathsf{output}_\ell(e)$. An output command is the only command that may cause a security violation. As such, the typing rule requires that both the value output and the decision to output are independent of any variables we are considering no longer tracking: both must have level $\bot$. Note that the security level ℓ of the output command is irrelevant: the type system is simply being used to determine whether any output command may be influenced by a variable we are considering no longer tracking.

Intuitively, if the judgment $\bot \vdash \overline{\bot}[x \mapsto \top]\,\{t'\}\,\Gamma$ holds for some Γ, then the value of variable x just before term t' executes does not influence any output that execution of t' may produce: it cannot affect either the decision to output or the value output. Thus, if t' is all that remains of the program to execute, then there is no need to track the security level of variable x in the rest of the program. More generally, if $\bot \vdash \overline{\bot}[X \mapsto \top]\,\{t'\}\,\Gamma$ holds, then none of the variables in the set X can cause a security violation, and there is no need to track the security level of any of them.

B. Monitor $\mathrm{M_{PERF}}$

We define a new monitor, $\mathrm{M_{PERF}}$, that uses this static analysis to reduce the number of variables that the monitor must track. Monitor $\mathrm{M_{PERF}}$ has a single inference rule, given in Figure 8. We write $\gamma \setminus X$ for the monitor environment that is undefined for variables X and otherwise behaves like monitor environment γ. Thus, $\mathrm{dom}(\gamma \setminus X) = \mathrm{dom}(\gamma) \setminus X$.

Intuitively, $\mathrm{M_{PERF}}$ takes a step $\langle \gamma, \sigma \rangle \xrightarrow[\mathrm{M_{PERF}}]{\alpha, t', m}_{\beta} \langle \gamma' \setminus X, \sigma' \rangle$ only if monitor $\mathrm{M_{RS}}$ takes a step to monitor configuration $\langle \gamma', \sigma' \rangle$, and variables $X \subseteq \mathrm{dom}(\gamma')$ cannot influence any output in the remainder of the program. Recall that the small step judgment for monitor configurations takes, in addition to the internal event, the original memory m and the term t' that will result if monitored execution is allowed to proceed one step. Thus, term t' is the remainder of the program.

```
1  // γ(x) = H,    γ(y) = L,    γ(z) = H
2  if z then
3        y := x;
4        output_L(x)
5  else
6        y := z;
7  output_L(1)
```

Figure 9. Example program

To illustrate the behavior of monitor $\mathrm{M_{PERF}}$, consider the example program in Figure 9. We assume a two-point lattice, $\{L, H\}$, where $L \sqsubseteq H$ and $H \not\sqsubseteq L$ and assume that at the beginning of the program variables x and z contain sensitive information (level H), and variable y contains non-sensitive information (level L). At line 2, the program branches on the value of z. Suppose the true branch is taken. The remainder of the program is then the term

$$t' \equiv \mathsf{y} := \mathsf{x};\ \mathsf{output}_\mathsf{L}(\mathsf{x});\ \mathsf{end};\ \mathsf{output}_\mathsf{L}(1).$$

Clearly variable z can no longer influence any output statement—it doesn't occur in t'—and so the monitor can stop tracking it. Similarly, variable y can no longer influence any output statement, and the monitor can stop tracking it, even though the assignment in line 3 would otherwise have raised the security level of y to H. Variable x may affect an output produced by t', so the monitor must continue to track its security level.

Suppose instead the false branch is taken. The remainder of the program is then the term $\mathsf{y} := \mathsf{z};\ \mathsf{end};\ \mathsf{output}_\mathsf{L}(1)$ and the monitor can immediately stop tracking all variables x, y, and z.

Figure 10 compares the monitor environments of $\mathrm{M_{RS}}$ and $\mathrm{M_{PERF}}$ after each line in the example program. We write $\varnothing$ for the monitor environment with an empty domain and write "—" when control flow does not reach the line.

C. Security

Monitor $\mathrm{M_{PERF}}$ enforces the same termination-insensitive noninterference security condition as $\mathrm{M_{RS}}$ (stated in Definition 1). Rather than show this directly, we prove that $\mathrm{M_{PERF}}$ is behaviorally equivalent to $\mathrm{M_{RS}}$: they allow exactly the same executions of the program.

Line #	$\mathrm{M_{RS}}$	$\mathrm{M_{PERF}}$	
		$\mathsf{z} \neq 0$	$\mathsf{z} = 0$
1	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto L, \mathsf{z} \mapsto H$	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto L, \mathsf{z} \mapsto H$	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto L, \mathsf{z} \mapsto H$
2	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto L, \mathsf{z} \mapsto H$	$\mathsf{x} \mapsto H$	$\varnothing$
3	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto H, \mathsf{z} \mapsto H$	$\mathsf{x} \mapsto H$	—
4	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto H, \mathsf{z} \mapsto H$	$\varnothing$	—
6	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto H, \mathsf{z} \mapsto H$	—	$\varnothing$
7	$\mathsf{x} \mapsto H, \mathsf{y} \mapsto H, \mathsf{z} \mapsto H$	$\varnothing$	$\varnothing$

Figure 10. Monitor state after executing each line of the example program given in Figure 9

First, we say that monitor M_1 is *at least as restrictive* as M_2 if for every execution that M_1 allows from some initial configuration, M_2 allows an execution from the same configuration with the same sequence of outputs.

Definition 2 (At least as restrictive)**.** *Monitor* M_1 *is* at least as restrictive *as* M_2 *if whenever*

$$\langle\langle t, m\rangle, \langle \gamma, \sigma\rangle\rangle \xrightarrow[\mathrm{M}_1]{\vec{\beta}}{}^{*} \langle\langle t', m'\rangle, \langle \gamma', \sigma'\rangle\rangle,$$

then there exists γ'' *and* σ'' *such that*

$$\langle\langle t, m\rangle, \langle \gamma, \sigma\rangle\rangle \xrightarrow[\mathrm{M}_2]{\vec{\beta}}{}^{*} \langle\langle t', m'\rangle, \langle \gamma'', \sigma''\rangle\rangle.$$

We say two monitors are behaviorally equivalent if they are both at least as restrictive as each other; that is, they allow exactly the same executions.

Definition 3 (Monitor behavioral equivalence)**.** *Monitors* M_1 *and* M_2 *are* behaviorally equivalent *if and only if* M_1 *is at least as restrictive as* M_2 *and* M_2 *is at least as restrictive as* M_1.

Monitors $\mathrm{M_{PERF}}$ and $\mathrm{M_{RS}}$ are behaviorally equivalent, which allows us to easily prove that $\mathrm{M_{PERF}}$ enforces the same security condition as $\mathrm{M_{RS}}$. Proofs of all results can be found in the accompanying technical report [23].

Lemma 1. $\mathrm{M_{PERF}}$ *is behaviorally equivalent to* $\mathrm{M_{RS}}$.

Proof (sketch): We use a standard noninterference proof to show that monitored execution under the two monitors is observationally equivalent. Interestingly, all outputs of the monitored execution are regarded as observable; we show that the variables our static analysis decides to stop tracking do not interfere with any of the outputs of the program, and thus both monitors make the same decisions regarding which outputs to allow. □

Theorem 2. *Monitor* $\mathrm{M_{PERF}}$ *is secure.*

Proof: Immediate from Lemma 1 and Theorem 1. □

D. Implementation issues

Cost of selectively tracking variables. Our static analysis identifies variables that the monitor can safely ignore. However, this provides a performance benefit only when it reduces the work that the monitor must perform. In a naïve implementation of monitor $\mathrm{M_{PERF}}$, selectively tracking variables may increase runtime overhead if the monitor is continually checking which variables to track.

We anticipate that performance is most likely to be improved for *inlined information-flow monitors* (e.g., [24, 25]), where the instrumented code is specialized for tracking just a subset of variables. For example, if the monitor needs to track variables y and z but not x, then no instrumentation is required for an assignment $\mathsf{x} := \mathsf{y} + \mathsf{z}$, thus removing the lookup of the security levels of y and z and the join operation, without requiring an explicit check to determine whether x should be tracked.

We are developing an inlined information-flow monitor with the ability to dynamically generate different versions of the same code that track different sets of variables. Because of the high overhead of dynamic code generation, this monitor will be most useful for applications in which sensitive data is infrequently introduced into the system. Executions will be lightly instrumented until sensitive information is introduced, at which time additional monitoring code will be generated to track information that may cause a security violation. The system can stop tracking variables (and return to the version of code with little instrumentation) once static analysis determines that a security violation can no longer occur. In a setting where sensitive data is frequently introduced, the benefit of reducing the number of variables that must be tracked may be less than the cost of generating several versions of code.

However, even for monitor implementations that cannot selectively start and stop tracking variables, or where the cost of selectively starting and stopping tracking variables is high, our static analysis may provide some performance benefits. If a variable can no longer affect whether a security violation occurs, it is sound to assign it any security level, as doing so will not change the behavior of the monitor. For some security lattices, join and comparison operations are less expensive for certain security levels. For example, if security levels are represented as partial functions (e.g., [26, 27]) then join and comparison operations can be more efficient on the partial function with an empty domain. Instead of removing variables from the domain of the monitor environment, the monitor can set the level of these variables to a level efficient for storage and computation.

On-the-fly analysis vs. pre-execution analysis. The operational semantics for monitor $\mathrm{M_{PERF}}$ implies on-the-fly static analysis, but our analysis can also be performed prior to

execution. Performing the static analysis on-the-fly allows its use with dynamic languages, in a similar manner to the use of on-the-fly static analysis by Askarov and Sabelfeld [12]. However, if used in this way, performing the static analysis on every execution step would most likely be too expensive. Instead, some subset of execution steps should perform the static analysis. This could be determined according to a schedule (e.g., every k steps), based on certain internal events (e.g., on every branch command and on the execution of dynamically-generated code, i.e., eval commands), or at program points identified by some static analysis.

If the static analysis is performed prior to execution, then the results of the analysis must somehow be communicated to the monitor. As we discuss above, we believe inlining the monitor and specializing the inlined code may be the most efficient way to take advantage of the static analysis results. However, other mechanisms are possible, such as the creation of a data structure that allows the monitor to look up results based on the remainder of the program to execute, perhaps represented by the current value of the program counter. Note that this data structure may contain just a subset of the analysis results; for example, only for program points where the set of variables to stop tracking is above some threshold size.

Implementing the analysis. The analysis is currently phrased as a syntax-directed type system that can check whether judgment $\bot \vdash \overline{\bot}[X \mapsto \top]\,\{t\}\,\Gamma$ holds for some set of variables X. However, to be useful, the analysis needs to *infer* the set of variables X. This corresponds to a *principal typing* problem [28], where given term t, we want to find a typing environment Γ such that $\bot \vdash \Gamma\,\{t\}\,\Gamma'$ holds, and Γ maps as many variables as possible to $\top$. Hunt and Sands [29] present a polynomial-time algorithm for inference of principal types for flow-sensitive security-type systems. Their results can be easily adapted for our setting, giving us an efficient algorithm to implement the analysis.

IV. Memory abstractions

In this section, we extend the hybrid information-flow monitor to a language with dynamically allocated memory and first-class references. The extended monitor is parameterized by a sound memory abstraction. Interestingly, not all sound memory abstractions are suitable for use in an information-flow monitor. We state sufficient conditions on the information-flow monitor and memory abstraction to enforce security (and informally describe necessary conditions). Many practical memory abstractions satisfy these sufficient conditions.

By being precise about the conditions for soundly incorporating a memory abstraction into an information-flow monitor, we allow monitor implementations to find an appropriate balance between efficiency and precision. In addition, these conditions highlight opportunities for the development of novel memory abstractions for information-flow control which use run-time information, including information about the state of the monitor, to improve precision.

$$
\begin{array}{lllll}
\text{Values} & v & ::= & \ldots \mid r & \\
\text{Expressions} & e & ::= & \ldots \mid *e & \\
\text{Commands} & c & ::= & \ldots \mid x := \mathsf{new}(e) \mid e_1 \leftarrow e_2 &
\end{array}
$$

$$\frac{e, m \Downarrow r \qquad m(r) = v}{*e, m \Downarrow v}$$

$$\frac{e, m \Downarrow v \qquad r \notin \mathrm{dom}(m)}{\langle x := \mathsf{new}(e), m\rangle \xrightarrow{\mathrm{new}(x,e,r)} \langle \mathsf{stop}, m[x \mapsto r][r \mapsto v]\rangle}$$

$$\frac{e_1, m \Downarrow r \qquad e_2, m \Downarrow v}{\langle e_1 \leftarrow e_2, m\rangle \xrightarrow{\mathrm{store}(e_1,e_2,r)} \langle \mathsf{stop}, m[r \mapsto v]\rangle}$$

Figure 11. Language extensions for dynamic memory

A. Language extensions

We extend the simple imperative language by adding the new syntactic forms and operational semantics rules given in Figure 11. Program configurations remain the same, although memories m now map both variables and *concrete locations* (or, simply, *locations*) to values. We use metavariable r to range over locations. Values in the language are now integers or locations.

Expression $*e$ evaluates e to a location r and looks up the contents of r in the current memory. Command $x := \mathsf{new}(e)$ creates a new location r, evaluates e to a value v, and updates the memory so that variable x maps to r and r maps to v. Internal event $\mathrm{new}(x, e, r)$ is generated when $x := \mathsf{new}(e)$ executes. Command $e_1 \leftarrow e_2$ evaluates e_1 to a location r, evaluates e_2 to a value v, updates the memory so that r maps to v, and issues internal event $\mathrm{store}(e_1, e_2, r)$.

The new language features enable new information flows. In addition to the information stored in locations, the choice of location accessed can be an information channel. When a pointer is dereferenced, the security level of the result depends on the security level of the pointer, as well as the security level of the value stored in the dereferenced location. Intuitively, a pointer dereference acts like a conditional, where the location accessed is conditional on the value of the pointer. To be sound, information-flow monitors must track these flows.

For example, consider the program in Figure 12. It allocates two locations, a and b, initialized to zero and one respectively. It then sets variable y to point to one of them based on sensitive information stored in variable h. The program contains three output commands. The security level of each output depends on the security level of both the pointer expression and the value stored in the location.

Line 6 outputs the contents of location a, which is always zero, reveals no sensitive information, and is thus secure. Line 7 outputs the contents of the location pointed to by

```
1  a := new(0);   b := new(1);
2  if (h > 0)   // variable h contains sensitive data
3        y := a
4  else
5        y := b;
6  output_L(*a); // safe output
7  output_L(*y); // unsafe output
8  y ← 2;
9  output_L(*a) // unsafe output
```

Figure 12. Example of information flow through pointer value

y. Which location y points to (and what value is output) depends on sensitive information. Thus, the security level of the value output is sensitive, and the output is insecure.

Line 9 occurs after the value 2 is stored into the location pointed to by y. One of the locations a or b is updated, depending on the sensitive value h. Regardless of which location is updated, the value stored in location a now depends on the sensitive value h, which makes the output insecure.

B. Sound memory abstractions

A *memory abstraction* for a program consists of a set of *abstract locations* and a function $\mathit{points\text{-}to}(\cdot,\cdot)$. During execution, concrete locations are allocated. Each concrete location is represented by one or more abstract locations, and each abstract location represents zero or more concrete locations. For a given expression e and location r, $\mathit{points\text{-}to}(e,r)$ is a set of abstract locations. Intuitively, soundness of a memory abstraction requires that given a set of expressions e that evaluate to the same concrete location r, there is at least one abstract location common to all of the expressions' points-to sets. Thus, the function $\mathit{points\text{-}to}(\cdot,\cdot)$ allows us to reason soundly about possible aliasing.

We say that expression e evaluates to location r during the execution of c if either $e, m \Downarrow r$ occurs during the execution of c or e is a variable x and judgment

$$\langle x := \mathsf{new}(e'), m\rangle \to \langle \mathsf{stop}, m[x \mapsto r][r \mapsto v]\rangle$$

occurs during the execution.

Definition 4 (Sound memory abstraction)**.** *A memory abstraction for program c is* sound *if for any execution of c and location r, if $\{e_1, \ldots, e_k\}$ is a set of expressions that all evaluate to location r during the execution of c, then*

$$\bigcap_{i=1}^{k} \mathit{points\text{-}to}(e_i, r) \neq \emptyset.$$

Note that a memory abstraction may ignore the location argument of $\mathit{points\text{-}to}(\cdot,\cdot)$ if, for example, the memory abstraction is generated statically. We include the location argument to allow memory abstractions that use runtime information. For presentation purposes, we provide $\mathit{points\text{-}to}(\cdot,\cdot)$ with only an expression and the location it evaluates to. This essentially restricts memory abstractions to flow-insensitive context-insensitive abstractions. We could generalize to allow flow-sensitive, context-sensitive, and even path-sensitive abstractions by providing additional arguments, such as the current program configuration, the current monitored program configuration, or a trace of program execution. For simplicity of presentation, we refrain from doing so.

NEW

$$\frac{\begin{array}{c} A = \mathit{points\text{-}to}(x,r) \\ \ell = \mathit{lev}(e,\gamma,m) \sqcup \mathit{lev}(\sigma) \\ \gamma'(s) = \begin{cases} \ell \sqcup \gamma(s) & s \in A \\ \gamma(s) & \text{otherwise} \end{cases} \end{array}}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{MEM}}]{\mathrm{new}(x,e,r),t,m} \langle \gamma'[x \mapsto \mathit{lev}(\sigma)], \sigma\rangle}$$

STORE

$$\frac{\begin{array}{c} A = \mathit{points\text{-}to}(e_1,r) \\ \ell = \mathit{lev}(e_1,\gamma,m) \sqcup \mathit{lev}(e_2,\gamma,m) \sqcup \mathit{lev}(\sigma) \\ \gamma'(s) = \begin{cases} \ell \sqcup \gamma(s) & a \in A \\ \gamma(s) & \text{otherwise} \end{cases} \end{array}}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{MEM}}]{\mathrm{store}(e_1,e_2,r),t,m} \langle \gamma', \sigma\rangle}$$

BRANCH

$$\frac{\mathit{lev}(e,\gamma,m) \sqcup \mathit{lev}(\sigma) = \ell}{\langle \gamma, \sigma\rangle \xrightarrow[\mathrm{M_{MEM}}]{\mathrm{branch}(e,c),t,m} \langle \gamma, (\ell, \textsc{Analyze}(c,m,\gamma,\ell)) : \sigma\rangle}$$

$$\mathit{lev}(e,\gamma,m) = \left(\bigsqcup_{x \in \mathrm{dom}(\gamma) \cap \mathrm{vars}(e)} \gamma(x) \right) \sqcup \bigsqcup_{a \in \mathrm{dom}(\gamma) \cap A} \gamma(a)$$

where

$$A = \bigcup \{ \mathit{points\text{-}to}(e', r) \mid *e' \text{ appears in } e \wedge (e', m \Downarrow r) \}.$$

Figure 13. Monitor with memory abstractions, $\mathrm{M_{MEM}}$

C. Monitor $\mathrm{M_{MEM}}$

We define a new monitor, $\mathrm{M_{MEM}}$, that can soundly track information flow for the language defined above. The monitor is parameterized on a sound memory abstraction and an analysis algorithm.

Monitor configurations for $\mathrm{M_{MEM}}$ remain unchanged, although the domain of monitor environments γ is extended to include abstract locations. Intuitively, $\mathrm{M_{MEM}}$ records for each abstract location a the level of information that may be learned by examining the contents of any of the concrete locations that a represents.

All small-step semantics inference rules for monitor $\mathrm{M_{RS}}$ are also inference rules for monitor $\mathrm{M_{MEM}}$, with the exception of the rule for internal event $\mathrm{branch}(e,c)$. Additional

inference rules for $\mathrm{M}_{\textsc{Mem}}$ are given in Figure 13.

When event $\mathrm{new}(x, e, r)$ (generated by allocation $x := \mathsf{new}(e)$) is encountered, the monitor updates the level of both the variable x and the abstract locations that represent the newly allocated location r. The level of x is set to the program counter level $lev(\sigma)$, since a pointer to the newly created reference reveals only that it was created. Abstract locations, on the other hand, are *weakly updated* to the join of the security level of expression e and the program counter level. Weak update is required because an abstract location may represent more than just one concrete location. As with other memory analyses, if it can be proved that an abstract memory location represents a single concrete location, strong update can be used (e.g., [16]).

The security level of expression e, $lev(e, \gamma, m)$, is the join of the levels of all variables that occur in e and the levels of all locations that might be dereferenced when e is evaluated. That is, if expression $*e'$ is a subexpression of e and $e', m \Downarrow r$, then location r will be dereferenced; the monitor is tracking the security level of those locations using abstract locations $\mathit{points\text{-}to}(e', r)$, and so $lev(e, \gamma, m)$ is at least as high as $\gamma(a)$ for all abstract locations a in the points-to set of e'.

Updating a location, $e_1 \leftarrow e_2$, generates event $\mathrm{store}(e_1, e_2, r)$. The monitor updates all abstract locations that represent a location that e_1 may evaluate to: $\mathit{points\text{-}to}(e_1, r)$. These locations are weakly updated with the join of the program counter level, the level of the value being stored ($lev(e_2, \gamma, m)$), and the level of the pointer to the updated location ($lev(e_1, \gamma, m)$), since *which* location is updated may reveal information.

We must also modify the rule for branches. Like monitor $\mathrm{M}_{\textsc{rs}}$, when $\mathrm{M}_{\textsc{Mem}}$ receives event $\mathrm{branch}(e, c)$, indicating the program has entered a branch guarded by expression e where c is the branch not taken, it pushes an analysis of the effects of c onto the monitor stack. However, in addition to possible updates to variables, the analysis must now also reason about possible updates to locations.

Since the specifics of this analysis may vary by memory abstraction, we parameterize our monitor with an analysis algorithm $\textsc{Analyze}(c, m, \gamma, \ell)$, which returns a monitor environment that approximates the environment that would result from the monitored execution of command c, the branch that was not taken.

In order for monitor $\mathrm{M}_{\textsc{Mem}}$ to soundly enforce security, the analysis algorithm must meet certain requirements. The key insight is that because the execution, or non-execution, of c depends on information at level ℓ, an entity that cannot observe information at level ℓ should not be able to distinguish between the monitor actually executing c (and updating the monitor configuration accordingly) and the monitor approximating the effect of executing c by using the analysis algorithm. This means that if executing c may change the security level of a variable or abstract location to ℓ or above, then $\textsc{Analyze}(c, m, \gamma, \ell)$ must return a monitor environment where the level of that variable or abstract location is also ℓ or above. This is a sufficient condition for $\mathrm{M}_{\textsc{Mem}}$ to soundly enforce security.

Definition 5 (Sufficient analysis algorithm)**.** *Analysis algorithm* $\textsc{Analyze}(c, m, \gamma, \ell_H)$ *is* sufficient *if for all* ℓ_L*,* m'*, and* σ *such that* $\ell_H \not\sqsubseteq \ell_L$*, and* $m =^{\ell_L}_{\gamma} m'$*, and* $lev(\sigma) = \ell_H$*, if*

$$\langle\langle c, m'\rangle, \langle\gamma, \sigma\rangle\rangle \rightarrow^* \langle\langle \mathsf{stop}, m''\rangle, \langle\gamma', \sigma\rangle\rangle$$

then

$$\textsc{Analyze}(c, m, \gamma, \ell_H) =_{\ell_L} \gamma'$$

where $\gamma_0 =_{\ell_L} \gamma_1$ *if and only if* $\mathrm{dom}(\gamma_0) = \mathrm{dom}(\gamma_1)$ *and* $\forall s \in \mathrm{dom}(\gamma_0).\ \gamma_0(s) \sqsubseteq \ell_L \vee \gamma_1(s) \sqsubseteq \ell_L \Rightarrow \gamma_0(s) = \gamma_1(s)$.

This property is sufficient but not necessary. To obtain a necessary condition, it must be weakened in two ways. First, instead of quantifying over all memories m' that are ℓ_L-equivalent to m, it is enough to quantify over memories that can be obtained by an execution of the program that, to an observer at level ℓ_L, appears equivalent to the execution that produced m. Second, monitor environments $\textsc{Analyze}(c, m, \gamma, \ell_H)$ and γ' do not need to be equal on every variable or abstract location that either environment maps to level ℓ_L or below. Instead, they need only agree on variables and abstract locations that could later affect the output of the program. We avoid stating the necessary property here due to the additional complexity of notation that would be required, and because the sufficient property described above is weak enough to use for all the memory abstractions we consider.

Theorem 3. *Monitor* $\mathrm{M}_{\textsc{Mem}}$*, when instantiated with a sound memory abstraction and sufficient analysis algorithm, is secure.*

Relationship of $\mathrm{M}_{\textsc{Mem}}$ to $\mathrm{M}_{\textsc{rs}}$ and $\mathrm{M}_{\textsc{Perf}}$. The function $\textsc{Analyze}(c, m, \gamma, \ell)$ is a generalization of the function $\textsc{Update}_\ell(c)$ used in $\mathrm{M}_{\textsc{rs}}$. Modulo providing $\textsc{Update}_\ell(c)$ with additional arguments (the memory m and the monitor environment γ), the sufficient conditions for $\textsc{Analyze}(c, m, \gamma, \ell)$ to make $\mathrm{M}_{\textsc{Mem}}$ soundly enforce security are also sufficient for $\textsc{Update}_\ell(c)$ to make $\mathrm{M}_{\textsc{rs}}$ soundly enforce security. This would allow for more sophisticated versions $\textsc{Update}_\ell(c)$. For example, an analysis could ignore the effect of dead code.

The selective tracking technique developed in Section III and used in $\mathrm{M}_{\textsc{Perf}}$ can also be used in $\mathrm{M}_{\textsc{Mem}}$ by extending the typing environments to also map abstract locations to security levels.

D. An example instantiation

To illustrate the use of our framework, we now describe a sound information-flow monitor based on a unification- or inclusion-based points-to analysis (e.g., [14, 13]). In this

memory abstraction, abstract locations are *allocation sites*—program points that create new locations. Each concrete location is represented by the abstract location corresponding to the allocation site at which it was created. The analysis computes points-to sets for each expression. If expression e evaluates to a concrete location r, then the allocation site of r is included in the points-to set of e, $\mathit{points\text{-}to}(e, r)$. Note that $\mathit{points\text{-}to}(e, r)$ ignores the second argument r, the concrete location to which e evaluates.

This memory abstraction satisfies Definition 4; if a set of expressions evaluate to the same concrete location r, then their points-to sets will all include the abstract location representing the allocation site of r, and thus have a non-empty intersection.

To complete the monitor, we define $\textsc{Analyze}(c, m, \gamma, \ell)$ as the natural generalization of $\textsc{Update}_\ell(c)$ in the presence of references. Function $\textsc{Analyze}(c, m, \gamma, \ell)$ returns a monitor environment γ' such that for every variable x, if c contains an assignment to x, then $\gamma'(x) = \ell$, otherwise $\gamma'(x) = \gamma(x)$; and for every statement $e_1 \leftarrow e_2$ in c, $\forall a \in \mathit{points\text{-}to}(e_1, \cdot)$, $\gamma'(a) = \ell$, otherwise $\gamma'(a) = \gamma(a)$. This is a sufficient analysis algorithm (Defintion 5) as it approximates the effects of the monitored execution of c. Interestingly, the set of variables and abstract locations for which $\textsc{Analyze}(c, m, \gamma, \ell)$ sets to security level ℓ is (and must be) exactly the set of variables and abstract locations that would be updated in the monitor environment during monitored execution of c.

E. Choosing a memory abstraction

No sufficient analysis algorithms. Surprisingly, there are sound memory abstractions for which there are no sufficient analysis algorithms. This shows that there are limits on which sound memory abstractions can be incorporated into secure information-flow monitors.

Consider a sound memory abstraction that for each concrete location r has an abstract location a_r and $\mathit{points\text{-}to}(e, r) = \{a_r\}$. A monitor using this memory abstraction is tracking information flow very precisely, on a per-location basis. However, no sufficient analysis algorithm exists. Consider a program where depending on sensitive information, branch c may or may not be taken. Command c performs some computation and, based on the result, decides to update one of two locations. To accurately approximate the effect of executing c, the analysis algorithm must determine which of the two locations is updated, which is undecidable, in general.

Novel memory abstractions. The statement of sufficient conditions for memory abstractions and analysis algorithms opens the possibility of developing novel memory abstractions that use security-relevant information to improve the precision and efficiency of information-flow monitoring.

For example, we can improve the precision of the example monitor in Section IV-D by tracking information flow by concrete location when updating a location in a context where both the program counter level and the pointer expression have security level $\bot_{\mathcal{L}}$. In this situation, the choice of location to update and the decision to update do not depend on sensitive information.

To achieve this, we extend the $\mathit{points\text{-}to}(\cdot, \cdot)$ function to take additional arguments: the current monitor configuration $\langle \gamma, \sigma \rangle$ and an argument write which is True only when $\mathit{points\text{-}to}$ is called to find the set of abstract locations for a concrete location that is being allocated or updated.

The new memory abstraction contains one abstract location for each concrete location. If $\mathit{write} = \mathsf{True}$ and the program counter level and the security level of the pointer expression e are both $\bot_{\mathcal{L}}$, then $\mathit{points\text{-}to}(e, r, \langle \gamma, \sigma \rangle, \mathit{write})$ returns $\{a_r\}$, the abstract location corresponding to the concrete location r; otherwise, it returns the set of abstract locations representing all concrete locations from an allocation site in the points-to set of e, as computed by the points-to analysis. This memory abstraction is sound and in some cases more precise than the memory abstraction presented in Section IV-D.

Function $\textsc{Analyze}(c, m, \gamma, \ell)$ for this monitor returns a monitor environment γ' such that if the program counter level $\mathit{lev}(\sigma)$ is $\bot_{\mathcal{L}}$, then $\gamma' = \gamma$. Otherwise, for every variable x, if c contains an assignment to x, then $\gamma'(x) = \ell$, otherwise $\gamma'(x) = \gamma(x)$; and for every statement $e_1 \leftarrow e_2$ in c, $\forall a \in \mathit{points\text{-}to}(e_1, \cdot, \cdot, \mathsf{False})$, $\gamma'(a) = \ell$, otherwise $\gamma'(a) = \gamma(a)$.

This analysis algorithm is sufficient and is able to track information flow precisely (i.e., at the granularity of single concrete locations), provided neither control flow nor the choice of which location to update depends on sensitive information. That is, explicit information flows can be tracked precisely.

Efficiency/precision tradeoffs. The memory abstraction used has a significant impact on the performance of an information-flow monitor. A more precise memory abstraction may have more abstract locations, which will increase both the storage required for monitor state and the complexity and number of security level updates.

The monitor we have defined supports a variety of memory abstractions and analysis functions. This allows us to consider trade-offs between efficient and precise memory abstractions with clear requirements for sound information-flow monitoring. At one extreme is a memory abstraction that maps all locations to a single abstract location. This will be sound, but very imprecise—a single piece of sensitive information stored in a location will irrevocably taint all memory. At the other extreme is the most precise sound memory abstraction, which is unusable in an information-flow monitor. Between these two extremes are many sound and useful memory abstractions.

For example, both unification- and inclusion-based pointer analyses (e.g., [14, 13]) are sufficient under our framework

but differ in precision and overhead. In a unification-based analysis, each allocation site belongs to a single points-to set. Thus, each points to set can be represented with a single abstract location. This is not the case for an inclusion-based analysis, which may be more precise, but at the expense of increasing both the number of abstract locations that must be tracked and the number of join operations on security levels. Shape analysis (e.g., [17]) is yet more precise, again in exchange for increased analysis complexity and runtime overhead. Our results allow all of these analyses and memory abstractions to be used soundly.

Efficient runtime representations. Some memory abstractions are more amenable to efficient representation at runtime. Some systems that use regions (e.g., [30]) or pool allocators (e.g., [31]) implicitly represent their abstract locations at run-time and can easily compute which abstract location(s) correspond to a given concrete location. An information-flow monitor can augment the data structures used to maintain regions and pools with security levels to efficiently track security state. An inlined information-flow monitor could further reduce overhead by directly inlining references to abstract locations where lookups are performed.

V. Related work

Russo and Sabelfeld [11] show the impossibility of sound, purely dynamic, flow-sensitive information-flow control. They also present a series of hybrid information-flow monitors, which combine dynamic and static analysis to provide sound flow-sensitive information control that is more precise than either purely static or purely dynamic techniques. Their monitors differ in behavior on insecure output: either stopping execution, suppressing output, or providing a default output. We extend their work by showing how additional static analysis can reduce the runtime overhead of information-flow monitors and show how a wide range of memory abstractions can be soundly incorporated into hybrid information-flow monitors.

Chandra and Franz [10] present a hybrid information-flow monitor for the Java Virtual Machine (JVM) that we believe is unsound. While they are careful to incorporate a sound pointer analysis into their approximation of untaken branches, on explicit updates they increase only the label of the object being modified. As a result, their monitor fails to control information flows through pointer dereference—they unknowingly trade soundness for precision. This highlights the importance of considering soundness while attempting to increase the precision of memory abstractions.

Le Guernic et al. [9] also present a flow-sensitive hybrid information-flow monitor which is subsumed by the monitors of Russo and Sabelfeld [11]. Le Guernic [32] extends this work to enforce noninterference in concurrent programs by ensuring the monitor prevents synchronization in program contexts with high-security program counter levels [33].

Shroff et al. [8] consider dynamic information-flow control in a language with dynamic memory allocation. Their system discovers dependencies within a program, either dynamically over several executions or statically. To deal with aliasing, their system must discover which dereferences may depend on which store updates, and in essence, hard-codes a particular pointer analysis. We show how to soundly incorporate a variety of memory abstractions, allowing a choice in the tradeoff between precision and efficiency.

Nair et al. [34] present Trishul, a hybrid system for information-flow control in the JVM that performs static analysis to determine which locations may be modified by code that is not executed and uses the results to soundly track implicit information flows. When the locations modified by code cannot be precisely determined, Trishul uses a *global taint* to conservatively approximate effects, essentially a single, coarse, abstract memory location.

Austin and Flanagan [6] consider sound purely-dynamic info flow tracking. They achieve soundness by requiring "no sensitive upgrade": non-sensitive memory locations cannot be upgraded to sensitive by assignment within a program context with a sensitive program counter level or by assignment via a sensitive pointer. They suggest modifying a program to preemptively upgrade non-sensitive locations that might otherwise require sensitive upgrade. This transformation results in similar precision to hybrid monitors that upgrade locations based on branches that could have, but were not, executed and is similar to the transformation implemented by Rifle [35]. They also introduce *sparse labeling*, where security labels are tracked explicitly only for data that migrates between information flow domains. They use sparse labeling to exploit *label locality*: the fact that items in a data structure tend to have the same security level. We believe that this complements our approach (i.e., tracking only items that may cause a security violation), and may lead to efficient representations of monitor environments in hybrid monitors.

Vachharajani et al. [35] propose Rifle, a system with architectural support for tracking information flow. Architectural support has the potential to improve the performance of information flow tracking, but is less portable than language-level approaches. Rifle tracks only explicit flows of information and handles implicit flows by performing a binary translation that makes implicit flows explicit, using a static analysis to reason about implicit information flows. No proof of soundness is given.

Newsome and Song [21] implement TaintCheck, which instruments binaries to track the flow of information within the program at byte-level granularity. Although they do not track implicit flows, they record detailed information about how data flows within the system. These detailed traces are analogous to a rich lattice of security levels. They report high overheads for their instrumented execution, particular of CPU-bound computation, and could possibly benefit from

our static analysis to reduce some of the instrumentation.

Tripp et al. [36] present Taint Analysis for Java (TAJ), a static analysis tool for detecting vulnerabilities in Java web applications. Their analysis does not consider implicit flows, but uses a novel technique of *hybrid thin slicing* to detect data dependencies of tainted sources.

Dynamic languages. Askarov and Sabelfeld [12] consider an information-flow monitor for *dynamic languages*, which can generate executable code at runtime. Their monitor uses on-the-fly static analysis of the dynamically generated code. Chugh et al. [37] consider information flow-control in JavaScript, a dynamic language, and perform a lightweight on-the-fly static analysis to determine whether dynamically generated code is secure; parameters for the on-the-fly static analysis are determined by a static analysis of the static portions of the code. Our results are applicable to dynamic languages, and our static analysis for selectively tracking variables can be performed on the fly.

Expressive language features. Russo and Sabelfeld [38] present a monitor for soundly tracking and controlling information flow due to *timeouts*, a mechanism for executing code snippets after a specified delay. Our selective tracking technique could be extended to this model by appropriately modifying the static analysis used. Russo et al. [39] precisely track and control information flow in dynamic tree structures. While some memory abstractions can reason quite precisely about tree structures, their monitor uses domain-specific knowledge and is thus likely more precise than ours, regardless of the memory abstraction used.

Inlining information-flow monitors. Chudnov and Naumann [24] prove that the information-flow monitor of Russo and Sabelfeld [11] can be inlined. Inlining enables compiler optimizations for the monitoring and facilitates incorporation of the monitor into existing systems. Magazinius et al. [25] present and prove sound a framework for inlining dynamic information-flow monitors that use the "no sensitive upgrade" mechanism to soundly control implicit information flows. Their framework permits on-the-fly inlining, thus providing support for dynamic languages. Venkatakrishnan et al. [40] present a program transformation that, in essence, inlines a hybrid information-flow monitor. They prove that the transformation enforces a noninterference-based security condition. Our work complements monitor inlining, and we believe the most benefit will be gained by applying our results to inlined monitors.

VI. Conclusion

We present two ways to use static analysis to increase the efficiency of hybrid information-flow monitors. First, we demonstrate a sound technique for selectively tracking variables during monitored program executions. Second, we derive sufficient conditions for soundly incorporating a variety of memory abstractions into a monitor for languages with dynamically allocated memory.

Selective tracking. Information-flow monitoring significantly decreases program performance. Part of this overhead is effort wasted on tracking security levels of data that cannot cause a security violation. We present a simple static analysis to soundly determine when variables can no longer influence dangerous operations and show that this analysis can be soundly incorporated into an information-flow monitor.

Memory abstractions. Practical information-flow control systems must deal with realistic language features, including dynamically allocated memory. The choice of memory abstraction used by an information-flow monitor has a large effect on both its precision and efficiency. While many information-flow control systems reason about memory, no clear requirements have been defined for permissible memory abstractions. We present sufficient conditions for incorporating memory abstractions and discuss how they apply to a variety of memory abstractions. This enables a principled exploration of tradeoffs between precision and efficiency, and opens the possibility of novel useful memory abstractions for information-flow monitors.

We are currently developing a system that dynamically generates instrumented code to enforce noninterference, guided by the results from the paper.

Acknowledgments

We thank the anonymous reviewers and Vikram Adve, John Criswell, Arushi Aggarwal, Will Dietz, and Gregory Malecha for helpful feedback on earlier versions of this paper. This research is sponsored by the Air Force Research Laboratory.

References

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[2] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[3] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, Jan. 1999, pp. 228–241.

[4] V. Simonet, "The Flow Caml System: documentation and user's manual," Institut National de Recherche en Informatique et en Automatique (INRIA), Technical Report 0282, Jul. 2003.

[5] J. S. Fenton, "Memoryless subsystems," *Computer Journal*, vol. 17, no. 2, pp. 143–147, May 1974.

[6] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of*

the 2009 Workshop on Programming Languages and Analysis for Security, 2009.

[7] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, 2009, pp. 352–365.

[8] P. Shroff, S. F. Smith, and M. Thober, "Dynamic dependency monitoring to secure information flow," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2007, pp. 203–217.

[9] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," *Proceedings of the 11th Annual Asian Computing Science Conference*, pp. 75–89, 2006.

[10] D. Chandra and M. Franz, "Fine-grained information flow analysis and enforcement in a Java virtual machine," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[11] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

[12] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *Proceedings of the IEEE Computer Security Foundations Symposium*, 2009.

[13] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994.

[14] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996, pp. 32–41.

[15] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, pp. 1–41, 2005.

[16] O. Lhoták and K.-C. A. Chung, "Points-to analysis with efficient strong updates," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011, pp. 3–16.

[17] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 3, pp. 217–298, 2002.

[18] M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and Computation*, vol. 132, no. 2, pp. 109–176, 1997.

[19] D. Gay and A. Aiken, "Memory management with explicit regions," in *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 1998, pp. 313–323.

[20] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[21] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[22] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, Jan. 2006, pp. 79–90.

[23] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," Harvard University, Tech. Rep. TR-05-11, Apr. 2011.

[24] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of the 23rd IEEE Security Foundations Symposium*, 2010.

[25] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," in *Proceedings of the IFIP International Information Security Conference*, 2010.

[26] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of the 16th ACM Symposium on Operating System Principles*. New York, NY, USA: ACM Press, 1997, pp. 129–142.

[27] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating system," in *Proceedings of the 20th ACM Symposium on Operating System Principles*. New York, NY, USA: ACM Press, Oct. 2005.

[28] T. Jim, "What are principal typings and what are they good for?" in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languagesg Languages*, 1996, pp. 42–53.

[29] S. Hunt and D. Sands, "From exponential to polynomial-time security typing via principal types," in *Proceedings of the 20th European Symposium on Programming*, 2011.

[30] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in Cyclone," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 2002, pp. 282–293.

[31] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*, October 2007.

[32] G. Le Guernic, "Automaton-based Confidentiality

Monitoring of Concurrent Programs," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, 2007, pp. 218–232.

[33] A. Sabelfeld, "The impact of synchronisation on secure information flow in concurrent programs," in *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, ser. Lecture Notes in Computer Science, vol. 2244. Springer-Verlag, 2002, pp. 225–239.

[34] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 3–16, February 2008.

[35] 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 *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, Dec. 2004.

[36] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009, pp. 87–97.

[37] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[38] A. Russo and A. Sabelfeld, "Securing timeout instructions in web applications," in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, 2009.

[39] A. Russo, A. Sabelfeld, and A. Chudnov, "Tracking information flow in dynamic tree structures," in *Proceedings of the European Symposium on Research in Computer Security*, 2009.

[40] V. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar, "Provably correct runtime enforcement of non-interference properties," in *Proceedings of the International Conference on Information and Communications Security*, 2006.