To appear in the *International Journal of Computer Mathematics*
Vol. 00, No. 00, Month 20XX, 1–24

Rewriting-Based Enforcement of Noninterference in Programs with Observable Intermediate Values

Afshin Lamei[a] and Mehran S. Fallah[b*]

[a,b]*Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), P.O.Box: 15875-4413, Tehran, Iran.*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

Program rewriting is defined as transforming a given program into one respecting some intended properties. This technique has recently been suggested as a means for enforcing security policies and proven more powerful than execution monitoring and static analysis. In this paper, we show that program rewriting can act as a sound and transparent mechanism to enforce information-flow policies in programs with observable intermediate values. To do so, we formulate progress-insensitive and progress-sensitive noninterference in the language presented in this paper. Then, we give rewriting mechanisms that correctively enforce the formulated policies. In fact, corrective enforcement is a realization of transparent rewriting in which the good behaviors of the program are preserved irrespective of whether it is secure or not. The proposed mechanisms make use of program dependence graphs. The nature of program rewriting, however, allows tracking those points on the graph that are actually traversed at run-time. This leads to more permissive mechanisms having little effect on the program's semantics.

Keywords: Information-flow security; noninterference; program dependence graphs; program rewriting; transparency.

2010 AMS Subject Classification: 68N15; 03D05; 03D10

1. Introduction

Program rewriting is defined as comprising those mechanisms that transform a given program so that the result satisfies some intended properties. This approach has traditionally been adopted for the migration of code between hardware platforms, instrumentation, and performance optimization [17]. Recently, it has been suggested as an effective method for enforcing *security policies* [46]. A program rewriter should be *sound* and *transparent* with respect to the given security policy. It is sound if the resulting code complies with the policy and transparent if the program's semantics is preserved.

A security policy can in general be defined as a subset of the power set of all executions, where an execution is an arbitrary sequence of states. That is, security policies are *hyperproperties* [13]. A program is also defined as a set of executions, i.e., all state sequences it may produce at run-time. A security policy can thus be thought of as the set of programs satisfying that policy. Some security policies are *properties* in the sense that they can be characterized as a set of executions. A property is indeed the power set of the set characterizing that property, and therefore, a program satisfies a security property if every possible execution of the program is in the set characterizing the property.

Many security policies, such as access control policies, are properties. There are, how-

*Corresponding author. Email: msfallah@aut.ac.ir

ever, important policies that cannot be expressed as properties. The Goguen-Meseguer noninterference [23], generalized noninterference [38], and observational determinism [56] are a few examples. *Noninterference* basically states that a low observer, who knows the program and can only observe public run-time values, can learn nothing about high (private) inputs to the program. This policy demands that public outputs should be the same in every pair of program executions that agree on public inputs. Thus, noninterference is not a property since it cannot be defined in terms of the individual executions satisfying it.

A principal question is 'which security policies can be enforced by rewriting?' In recent years, a great deal of research has been dedicated to the classification of enforceable security policies [5–7, 28, 33, 34, 36, 45**?**]. Despite endeavors made, important aspects of the formal characterization of program rewriting have remained open. At the same time, recent works evidence the capacity of program rewriting to enforce various security properties [21, 25, 41, 51, 54]. The proposed solutions rewrite the code by in-lining a truncation or edit automaton [**?**]. An in-lined truncation automaton acts as a reference monitor [20]. It takes security-relevant run-time events as inputs and halts the execution whenever a policy-violating event is about to occur. An edit automaton [32] may additionally add new actions to the execution and may condition some others on specific events in the future. Nonetheless, program rewriting is beyond just using in-lined reference monitors and edit automata and can conceivably be employed as a means for enforcing policies that are not properties. While remaining transparent, a program rewriter can make any changes to the code.

In this paper, we aim at showing the applicability of program rewriting, as superior to static checking and execution monitoring [26], to the enforcement of two important information-flow policies that are not properties. The investigated policies are *progress-insensitive* and *progress-sensitive* noninterference [2]. These policies reflect what is expected of programs having interaction with environment during run-time. In progress-insensitive noninterference, it is assumed that low observers can only see intermediate low outputs. A low observer in the progress-sensitive formulation of noninterference can additionally observe the *progress status* of the program in the sense that he can draw a distinction between program divergence and the situation in which the program has terminated or is computing the next observable value. Prior work on information-flow security, e.g., [18, 42, 43], mainly tries to enforce a variant of progress-insensitive noninterference. There are, however, few solutions for progress-sensitive noninterference [11, 40, 48, 57]. They are static mechanisms that are excessively restrictive and reject many healthy programs only due to the existence of control-flow dependencies on high values.

The program rewriters proposed in this paper make use of program dependence graphs (PDGs) [22, 29], which are proven at least as powerful as security type systems in detecting potential information flows [24, 37, 55]. Unlike earlier information flow control mechanisms, our rewriters preserve valid behaviors of programs, i.e., those program executions that do not lead to a security violation. This is due to the way we define and enforce transparency, the most challenging requirement of an effective enforcement mechanism. The rewriters proposed in this paper modify the code so that explicit and implicit illegal flows as well as the ones arising from termination channels are prevented at run-time. They also make use of path conditions to more accurately verify the conditions required for an illegal flow to actually occur, thereby preserving more valid behaviors of the given program. To deal with transparency, we formalize the concept of *corrective enforcement* for our formulations of noninterference—it is indeed an extension of the same concept for security properties [27]. A rewriter correctively enforces a policy if the

valid behaviors of the input program are preserved irrespective of whether the program is secure or not. We prove that the proposed rewriters are sound, that is, they definitely produce secure programs. In brief, this paper has the following contributions.

- Unlike many earlier security mechanisms, the ones proposed in this paper do not reject, before or during run-time, a policy-violating program. They are rewriters and convert insecure programs to secure ones.
- This paper gives a formalization of progress-insensitive and progress-sensitive noninterference in programs with intermediate observable values. This formalization is based on the semantics of the *while language* (WL) presented in this paper.
- The proposed rewriters are provably sound and transparent in enforcing noninterference. Transparency is also defined in such a way that the set of possible executions of the transformed program is as close as possible to that of the input program whether it is secure or not.

We proceed as follows: Section 2 is an overview of related works. Section 3 gives basic definitions and some motivating examples of rewriting as a security mechanism. Section 4 gives a formalization of progress-insensitive and progress-sensitive noninterference in the language introduced in the same section. Section 5 first sketches out how noninterference can be enforced by a rewriting mechanism based on program dependence graphs. Then, it elaborates on rewriting for the two formulations of noninterference given in Section 4. Section 6 is on the soundness and transparency of our rewriting mechanisms. Section 7 concludes the paper.

2. Related Work

Recent results in enforcing information-flow policies by means of purely dynamic or hybrid mechanisms indicate the capacity of program transformers for doing so. RIFLE [52] is an assembly-level rewriting mechanism to track implicit and explicit flows at run-time. Beringer [9] generalizes the idea of RIFLE and gives a formalism so that it can be proven that the proposed mechanism soundly enforces flow-sensitive multilevel security in a while language. Our rewriting mechanisms differ from RIFLE and its extension in many respects. In particular, instead of flow-sensitive typing, our static analysis is based on program dependence graphs. More importantly, dynamic updating of the security class of a value is not allowed in our mechanisms. This prevents the leakage arising from removing a value from the purview of a low observer as a result of updating the security class of that value.

Venkatakrishnan et al. [53] propose a hybrid transformation method to enforce noninterference. The transformed program tracks the security levels of assignments and terminates whenever an illegal flow is about to occur. In this way, it is only applicable to those formulations of noninterference disregarding the termination behavior of programs. Magazinius et al. [35], devise a framework for in-lining dynamic security monitors while the program is being executed. The method guarantees termination-insensitive noninterference and can be applied to languages such as Perl and Javascript that support dynamic code evaluation. It also requires the transformer to be available at run-time so that an appropriate monitor can be in-lined in the code which is dynamically generated. Chudnov and Naumann [12] propose a hybrid monitor for flow-sensitive noninterference—the concept of noninterference adapted for systems in which security classes can be updated dynamically—in languages with dynamic code evaluation. Indeed, their mechanism stores and processes information such as shadow variables, labels, variables inside branches and

so on at run-time similar to the VM-monitor proposed by Russo et al. [42]. This may lead to an unacceptable run-time overhead. The mechanism does not inhibit termination channels either. Santos and Rezk [44] extend the mechanism into a core of Javascript.

It has been proven that there is no purely dynamic mechanism to enforce flow-sensitive noninterference [42]. This has led to proposals that put some syntactic restrictions on the code, make use of static information in monitoring, or even leverage on multiple executions of programs. Bello and Bonelli [8] suggest an execution monitor using a run-time dependency analysis [47]. In this way, they can go beyond policies such as *no-sensitive* [3] and *permissive* [4] *upgrade*. To detect an illegal flow, however, their proposal, and some other similar solutions [19], may require several runs of the program, a thing which is not possible in many applications. Le Guernic et al. [31] design an automaton that receives abstract events at run-time and edits the execution using some static information. The mechanism is interesting but allows termination channels.

3. Preliminaries

A program can be modeled as a set of traces ψ which is a subset of the universe of all traces Ψ. Each trace in ψ represents one of the possible executions of the program and is considered as a finite or an infinite sequence of states. A state itself is a mapping of variables to values. For a given trace $t = \sigma_0\,\sigma_1\,...$, the symbols $t[i]$, $t[..i]$, and $t[i..]$ stand for σ_i, $\sigma_0\,...\,\sigma_i$, and $\sigma_i\,\sigma_{i+1}...$, respectively. It is worth noting that sometimes it is more convenient to model a program by the way it interacts with environment. To do so, we may model a program execution as a sequence of input/output events raised in transit from one state to the other. In this way, an execution in the form of a sequence of states can be converted into a sequence of input/output events. In defining security policies, we usually consider a mapping from events to a lattice of security classes. For the sake of simplicity, it is often assumed that there are only two security classes, *high* (private) and *low* (public).

A policy is defined to be a subset of the power set of the universe of all traces Ψ. That is, every policy is a hyperproperty and the policy space $\mathcal{P}$ is

$$\mathcal{P} = 2^{2^{\Psi}}, \tag{1}$$

where 2^X denotes the power set of set X. A program represented by $\psi \subseteq \Psi$ satisfies policy P, written $P(\psi)$, if $\psi \in P$. A policy $P \in \mathcal{P}$ is called a property if $P = 2^{\psi}$ for some $\psi \subseteq \Psi$.

A program rewriter RW for a given (security) policy $P \in \mathcal{P}$ transforms programs and is characterized as a total function $RW : 2^{\Psi} \to 2^{\Psi}$. A program rewriter RW for policy P is sound if

$$\forall \psi \subseteq \Psi.\, P(RW(\psi)). \tag{2}$$

It is also said to be transparent if

$$\forall \psi \subseteq \Psi.\, P(\psi) \Rightarrow \psi \approx RW(\psi), \tag{3}$$

where '$\approx$' denotes a behavioral equivalence relation over programs [26].

The concept of transparency defined in (3) lacks an important feature that seems essential to program rewriters. It puts no restriction on the programs not complying

with the policy. We believe that there should be a relation, not necessarily an equivalence relation, between a program and its rewritten version even though the input program does not satisfy the policy. Such a relation captures the idea that changes to a program should be minimal in the sense that the set of possible executions of the transformed program are as close as possible to that of the input program. There is a similar conception of transparency for execution monitors, termed corrective enforcement, which stipulates that valid parts of any execution should be preserved in the corresponding transformed execution [10, 27]. However, execution monitors can only enforce properties, a specific kind of policy, by monitoring and transforming individual executions. Thus, the concept of corrective enforcement should be revised and adapted for program rewriting.

Our formulation of transparency involves a preorder $\sqsubseteq$ on programs. This relation is defined in terms of the good features of programs. Indeed, we first decide on an appropriate abstraction function $\mathcal{A} : 2^{\Psi} \to \mathcal{I}$, which captures some particular features of programs, and a preorder $\preceq$ on the abstract values returned by $\mathcal{A}$. The preorder $\sqsubseteq$ is then defined as

$$\forall \psi, \psi' \subseteq \Psi.\ \psi \sqsubseteq \psi' \Leftrightarrow \mathcal{A}(\psi) \preceq \mathcal{A}(\psi'). \tag{4}$$

A program rewriter is said to be transparent with respect to $\sqsubseteq$ if

$$\forall \psi \subseteq \Psi.\, \psi \sqsubseteq RW(\psi). \tag{5}$$

A transparent rewriter should indeed produce a secure program that is higher than, or equal to, the input program on $\sqsubseteq$. This is an extension of the notion of *corrective*$_{\sqsubseteq}$ enforcement, in the literature of execution monitoring [28], to program rewriting.

Now, we give examples of how noninterference, a security policy which is not a property, may be enforced through program rewriting. Noninterference basically states that variations in high inputs to a program should not be reflected in its low outputs. That is, any two inputs that are equivalent in the view of a low observer should produce outputs being indistinguishable to that observer. The problem of identifying the programs satisfying noninterference is, in general, undecidable. Therefore, noninterference cannot generally be enforced by static mechanisms—this is the reason why the type systems already proposed for noninterference are conservative and may reject secure programs. It is not co-recursively enumerable either. Thus, it cannot be enforced by execution monitors that monitor a running program to infer if it violates noninterference [26]. Program rewriting does not make changes to the very nature of noninterference. Instead, it produces a new program that satisfies noninterference possibly at the cost of changes it may impose on the behavior of the original program. When such changes approach to the least required for noninterference, its advantages over static and monitoring mechanisms will appear.

The first example is the *secret file policy* (SFP). According to SFP, low users should not be able to obtain information about the presence of secret files in a directory by issuing the directory listing command. To enforce this policy, a naive solution is to block any request for such a listing if the target directory contains secret files. This solution, however, opens a new channel in which the denial of a request means that there are secret files in the directory. In fact, it cannot provide information-flow security in terms of noninterference—there is a wide variety of noninterference formulations, each giving a specific meaning to the absence of flow from confidential to public information receptacles in a program, e.g., [14, 18, 43]. More precisely, SFP is an instantiation of noninterference in a file system that stores the files provided by the users in directories and returns the

list of files in a directory when requested. A file system M satisfies SFP if the following holds for the set ψ_M of its traces, where $t_{l_{in}}$ and $t_{l_{out}}$ are the sequences of low inputs and low outputs generated by trace t, respectively.

$$\forall t, t' \in \psi_M.\ t_{l_{in}} = t'_{l_{in}} \ \Rightarrow\ t_{l_{out}} = t'_{l_{out}}. \tag{6}$$

Here, an input either puts one or more files in a directory or is a request for the list of files in a directory. The only output is the list of files shown to the user whose listing command is the last input. The system introduced above does not satisfy (6). Given that two programs are equivalent if they generate equivalent trace sets, (6) is neither decidable nor co-recursively enumerable. Therefore, it is not enforceable by static or purely dynamic mechanisms. Instead, if we rewrite the code so that the files labeled secret are removed from the list, the result respects SFP. This suggests the superiority of program rewriting to static and purely dynamic mechanisms.

As another example, consider a program that receives requests for a shared resource, allocates the resource to the first request, and denies other requests until the resource is released. This program does not satisfy noninterference. If the first request is from a high user, low users can learn something about high inputs. This program can be rewritten in such a way that a high user's access to the resource is transferred to the low user who makes the request for the same. In this way, what is observable to low users does not depend on high inputs.

In both examples above, the rewriter produces a new system in which low outputs are not affected by the inputs from high users. Information flow from high to low may be explicit such as assigning a high value to a low variable, implicit such as the flow from high to low when the value of a low variable is conditioned on a high value, or even through the timing and termination behavior of the program. In general, depending on the capabilities of low users, also known as the *attacker model*, there are many subtle ways for information flow from high to low. This is where things become interesting. A rewriting mechanism for a specific formulation of noninterference should take into account all possible kinds of illegal flows reflected by the underlying attacker model.

4. Security

We give a formalization of security in terms of noninterference for programs with intermediate outputs. In doing so, we first specify the syntax and semantics of a *while* language, WL, which allows programs to output values any time during run-time. This is in contrast to many works on information-flow security where the output of a program is considered to be its final state. Then, we formulate two views of noninterference, progress-insensitive and progress-sensitive, in the form of equivalence relations on program executions.

4.1 *WL: A while language*

The abstract syntax of WL is shown in Figure 4.1. An expression is a constant integer or Boolean, an integer variable, or a binary operation or relation on expressions. The set of commands is comprised of 'Nop' for no operation, '$x = exp$' for assignment, '$in_\Gamma\ varlist$' and '$out_\Gamma\ x$' for input and output where Γ is a security level, '$c; c$' for sequencing, conditionals, and '*while*' for creating loops. There is also a certain output command '$out_\Gamma\ \bot$' that outputs the constant $\bot$ differing from all other constants of the language. The most important control structure of WL is *while*, hence the name.

$$
\begin{aligned}
exp &::= b \mid n \mid x \mid exp_1\ op\ exp_2 \\
c &::= Nop \mid x = exp \mid in_\Gamma\ varlist \mid out_\Gamma\ x \mid out_\Gamma \perp \mid c; c \\
&\quad \mid if\ exp\ then\ c\ endif \mid if\ exp\ then\ c\ else\ c\ endif \\
&\quad \mid while\ exp\ do\ c\ done \\
op &::= == \mid < \mid > \mid <= \mid + \mid - \mid \vee \mid \wedge \\
varlist &::= x \mid x,\ varlist
\end{aligned}
$$

Figure 1. Syntax of WL.

We assume a batch of inputs at the beginning of the code. Outputs, however, may occur at any point. A trace is a sequence of states where transition to a state may be accompanied by an event. Thus, a program execution can be represented by the sequence of events it produces. In fact, events are inputs from and outputs to the environment due to *in* and *out* commands. Silent divergence is also considered as an event, shown by $\circlearrowleft$, in case it is observable to low users. The set of all possible sequences of events a program M may generate is denoted by $\mathcal{S}(M)$. Figure 2 is an example program and all sequences of events it may generate where it is assumed that variables only take 0 or 1. For a value v, $\check{v}_\Gamma$ and $\hat{v}_\Gamma$ are input and output events of security level Γ, respectively. Similarly, for a sequence of events $S \in \mathcal{S}(M)$, we use $\check{S}_\Gamma$, $\hat{S}_\Gamma$, and S_Γ to refer to input, output, and the entire subsequence of S at security level Γ.

As $\mathcal{S}(M)$ reflects all possible behaviors of program M, one can investigate it to check that M respects a given security policy. For instance, our example program in Figure 2 does not satisfy the noninterference defined by (6), because the subsequence $\langle \check{0}_L, \check{0}_L, \hat{1}_L \rangle$ is not consistent with the high input $\langle \check{0}_H \rangle$, for example. That is, a low observer can exclude the possibility of high input $\check{0}_H$ whenever he observes the sequence $\langle \check{0}_L, \check{0}_L, \hat{1}_L \rangle$ of low events. The subsequence $\langle \check{0}_L, \check{1}_L, \hat{0}_L \rangle$ precludes $\langle \check{1}_H \rangle$ as well. In essence, the low outputs produced at line 7 are influenced by the high input taken at line 2. From now on, we also use the term trace to refer to the sequence of events generated by a sequence of states.

1. $in_L\ l_1, l_2$;
2. $in_H\ h_1$;
3. $if\ \ (l_1 == 0)\ \ then$
4. $\quad l_2 = h_1$;
5. $else\ \ Nop$;
6. $endif$
7. $out_L\ l_2$;

(a)

$\langle \check{0}_L, \check{0}_L, \check{0}_H, \hat{0}_L \rangle$
$\langle \check{0}_L, \check{0}_L, \check{1}_H, \hat{1}_L \rangle$
$\langle \check{0}_L, \check{1}_L, \check{0}_H, \hat{0}_L \rangle$
$\langle \check{0}_L, \check{1}_L, \check{1}_H, \hat{1}_L \rangle$
$\langle \check{1}_L, \check{0}_L, \check{0}_H, \hat{0}_L \rangle$
$\langle \check{1}_L, \check{0}_L, \check{1}_H, \hat{0}_L \rangle$
$\langle \check{1}_L, \check{1}_L, \check{0}_H, \hat{1}_L \rangle$
$\langle \check{1}_L, \check{1}_L, \check{1}_H, \hat{1}_L \rangle$

(b)

Figure 2. A program and its traces. (a) Program M. (b) The trace set $\mathcal{S}(M)$ where $\check{v}_\Gamma$ and $\hat{v}_\Gamma$ denote input and output values $v \in \{0, 1\}$ at security level $\Gamma \in \{L, H\}$.

Formal semantics of WL bears on how a program produces a trace of events at runtime. As shown in Figure 3, it builds on configurations where a configuration is either a

$$\frac{}{(\mathbf{Nop}, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S})}\ \textsc{Nop1}$$

$$\frac{}{(x = exp, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma[x \mapsto \sigma(exp)], \check{S}, \hat{S})}\ \textsc{Assign}$$

$$\frac{\Gamma \downarrow v}{(in_\Gamma\ x, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma[x \mapsto v], \check{S} \cdot \langle \check{v}_\Gamma \rangle, \hat{S})}\ \textsc{InputVar}$$

$$\frac{\Gamma \downarrow v}{((in_\Gamma\ x,\ varlist), \sigma, \check{S}, \hat{S}) \longrightarrow (in_\Gamma\ varlist, \sigma[x \mapsto v], \check{S} \cdot \langle \check{v}_\Gamma \rangle, \hat{S})}\ \textsc{InputVarList}$$

$$\frac{\sigma(x) = v}{(out_\Gamma\ x, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S} \cdot \langle \hat{v}_\Gamma \rangle)}\ \textsc{OutputVar}$$

$$\frac{}{(out_\Gamma\ \bot, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S} \cdot \langle \hat{\bot}_\Gamma \rangle)}\ \textsc{Output}\bot$$

$$\frac{(c_1, \sigma, \check{S}, \hat{S}) \longrightarrow (c_1', \sigma', \check{S}', \hat{S}')}{(c_1; c_2, \sigma, \check{S}, \hat{S}) \longrightarrow (c_1'; c_2, \sigma', \check{S}', \hat{S}')}\ \textsc{Seq1}$$

$$\frac{}{(\epsilon; c_2, \sigma, \check{S}, \hat{S}) \longrightarrow (c_2, \sigma, \check{S}, \hat{S})}\ \textsc{Seq2}$$

$$\frac{\sigma(exp) = \text{true}}{(\mathbf{if}\ exp\ \mathbf{then}\ c\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (c, \sigma, \check{S}, \hat{S})}\ \textsc{If-True}$$

$$\frac{\sigma(exp) = \text{false}}{(\mathbf{if}\ exp\ \mathbf{then}\ c\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S})}\ \textsc{If-False}$$

$$\frac{\sigma(exp) = \text{true}}{(\mathbf{if}\ exp\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (c_1, \sigma, \check{S}, \hat{S})}\ \textsc{If-Else-True}$$

$$\frac{\sigma(exp) = \text{false}}{(\mathbf{if}\ exp\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (c_2, \sigma, \check{S}, \hat{S})}\ \textsc{If-Else-False}$$

$$\frac{}{(\mathbf{while}\ exp\ \mathbf{do}\ c, \sigma, \check{S}, \hat{S}) \longrightarrow (\mathbf{if}\ exp\ \mathbf{then}\ (c; \mathbf{while}\ exp\ \mathbf{do}\ c)\ \mathbf{else}\ \mathbf{Nop}, \sigma, \check{S}, \hat{S})}\ \textsc{While}$$

Figure 3. Small-step semantics for WL.

terminal configuration $(\epsilon, \sigma, \check{S}, \hat{S})$ or an intermediate one $(c, \sigma, \check{S}, \hat{S})$ where ϵ denotes an empty string and c is a nonempty string of terminal symbols. The set of all configurations is denoted by $\mathcal{C}$. In fact, semantic rules define the small-step transition relation '$\longrightarrow$' on $\mathcal{C}$ where expressions are interpreted as usual arithmetic and Boolean expressions. The judgment $(c, \sigma, \check{S}, \hat{S}) \longrightarrow (c', \sigma', \check{S}', \hat{S}')$ means that the execution of command c in state σ and in a configuration with input and output traces $\check{S}$ and $\hat{S}$ results in the configuration $(c', \sigma', \check{S}', \hat{S}')$. A transition may generate no event, that is, $\check{S}' = \check{S}$ and $\hat{S}' = \hat{S}$. When a transition produces an event, either $\check{S}' = \check{S} \cdot \langle e \rangle$ or $\hat{S}' = \hat{S} \cdot \langle e \rangle$ where e is the input or output event generated by this transition and '$\cdot$' is the concatenation operator. We also use '$\longrightarrow^*$' as the transitive closure of '$\longrightarrow$'.

Let X be the set of variables. A state σ is defined to be a mapping $\sigma : X \to \mathbb{N} \cup \{\text{null}\}$, where $\mathbb{N}$ is the set of all integers an 'null' is the value of variables before being assigned with inputs from the environment. The value of x in σ is $\sigma(x)$ and $\sigma[x \mapsto v]$ is a state obtained from σ by updating the value of x to v. The value of an expression exp in σ is also denoted by $\sigma(exp)$. By '$\Gamma \downarrow v$', we mean that the next input the environment provides is v at security level Γ. For the sake of brevity, we do not include the rules for updating the environment.

4.2 *Noninterference*

Since WL has output commands, intermediate steps of computations should be taken into account in the formulation of noninterference. That is, an effective security requirement for WL programs is in the form of a progress-insensitive or progress-sensitive noninterference [2]. In order to formally specify such requirements, we introduce some concepts.

If $(c, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma', \check{S}', \hat{S}')$, we say that command c terminates in one step. Moreover, $(c, \sigma, \check{S}, \hat{S}) \Rightarrow (c', \sigma', \check{S}', \hat{S}')$ denotes evaluation until an observable event where $c' \neq \epsilon$ and $\hat{S}' = \hat{S} \cdot \langle \hat{v} \rangle$. Likewise, $(c, \sigma, \check{S}, \hat{S}) \Rightarrow (\epsilon, \sigma, \check{S}', \hat{S}')$ denotes termination with or without any observable events—an observable event is created when an output command is executed. The judgment $(c, \sigma, \check{S}, \hat{S}) \Rightarrow (c', \sigma', \check{S}', \hat{S}')$ is defined by the rules in Figure 4.

$$\frac{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \longrightarrow^* (c_2, \sigma_2, \check{S}_2, \hat{S}_2) \quad \hat{S}_2 = \hat{S}_1 \cdot \langle \hat{v}_\Gamma \rangle \quad c_2 \neq \epsilon}{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Rightarrow (c_2, \sigma_2, \check{S}_2, \hat{S}_2)} \textsc{EvlObs}$$

$$\frac{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \longrightarrow^* (\epsilon, \sigma_2, \check{S}_2, \hat{S}_2)}{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Rightarrow (\epsilon, \sigma_2, \check{S}_2, \hat{S}_2)} \textsc{Term}$$

Figure 4. Rules defining the relation '$\Rightarrow$'.

A program *diverges silently* if it does not terminate nor does evaluate to any observable event. In formal terms, $(c, \sigma, \check{S}, \hat{S})$ diverges silently if

$$\nexists C' \in \mathcal{C}.\ (c, \sigma, \check{S}, \hat{S}) \Rightarrow C'.$$

An attacker (low observer) may or may not be able to observe silent divergence of programs.

For two traces S and S', we say that S is a *prefix* of S', or S' *extends* S, if there exists S'' such that $S' = S \cdot S''$. In such a case, S'' is denoted by $S' \setminus S$. Now, we define the *evaluation relation* '$\Downarrow$' by the rules in Figure 5. As programs may produce infinitely many observable events, the rules are interpreted *coinductively*.

$$\frac{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Rightarrow (c_1', \sigma_1', \check{S}_1', \hat{S}_1') \quad (c_1', \sigma_1', \check{S}_1', \hat{S}_1') \Downarrow \hat{S}_0 \quad c_1' \neq \epsilon}{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Downarrow (\hat{S}_1' \setminus \hat{S}_1) \cdot \hat{S}_0} \textsc{Ev1}$$

$$\frac{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Rightarrow (\epsilon, \sigma_1', \check{S}_1', \hat{S}_1')}{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Downarrow \hat{S}_1' \setminus \hat{S}_1} \textsc{Ev2}$$

$$\frac{\nexists C \in \mathcal{C}.\ (c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Rightarrow C}{(c_1, \sigma_1, \check{S}_1, \hat{S}_1) \Downarrow \langle \circlearrowleft \rangle} \textsc{SilEv}$$

Figure 5. Rules defining the evaluation relation '$\Downarrow$'.

Definition 1 *Two output traces $\hat{S}_1$ and $\hat{S}_2$ are said to be equivalent in the view of low observers, written $\hat{S}_1 =_L \hat{S}_2$, if their subsequences of all low events—$\circlearrowleft$ and those resulting from out_L commands—are equal.*

We also define *progress-insensitive equivalent* traces as two output traces that are equivalent in the view of low observers up to the first point at which divergence appears in one of the traces. In other words, the sequence of low events before divergence in one trace should be a prefix of that in the other trace. A low observer cannot differentiate between two progress-insensitive output traces if he cannot draw a distinction between program divergence and the situation in which the program has been terminated or is computing the next observable value.

DEFINITION 2 *Two output traces $\hat{S}_1$ and $\hat{S}_2$ are said to be progress-insensitive equivalent, noted $\hat{S}_1 \sim_{PINI} \hat{S}_2$, iff*

$$\big(\hat{S}_1 =_L \hat{S}_2\big) \vee \exists \hat{S}, \hat{S}', \hat{S}''.\ \Big(\big(\hat{S} =_L \hat{S}'\big) \wedge \Big(\big(\hat{S}_1 \setminus \hat{S} = \langle \circlearrowleft \rangle \wedge \hat{S}_2 \setminus \hat{S}' = \hat{S}''\big) \vee \big(\hat{S}_2 \setminus \hat{S} = \langle \circlearrowleft \rangle \wedge \hat{S}_1 \setminus \hat{S}' = \hat{S}''\big)\Big)\Big). \quad (7)$$

Progress-insensitive noninterference, PINI for short, stipulates that any two executions of the program with input sequences that are equivalent in the view of low observers should result in progress-insensitive equivalent output traces. Our formal definition of progress-insensitive noninterference is based on some assumptions and notations. It is assumed that any WL program begins with a batch of consecutive input commands which we call it the input block and that no input command appears outside this block. A command in the input block assigns the values provided by the environment to variables in that command. The security level of such values is assumed to be that of the input command. In this way, each variable in the input block can be thought of as a high or a low variable. The sets $\mathcal{H}$ and $\mathcal{L}$ denote high and low variables in the input block, respectively. The program obtained from M by removing its input block is denoted by M^c. Moreover, two states σ and σ' are called low-equivalent, written $\sigma =_L \sigma'$, if $\sigma(l) = \sigma'(l)$ for any $l \in \mathcal{L}$.

DEFINITION 3 *Program M satisfies progress-insensitive noninterference if for any two configurations $m = (M^c, \sigma, \check{S}, \lambda)$ and $m' = (M^c, \sigma', \check{S}', \lambda)$, $\sigma =_L \sigma'$ implies $\hat{S} \sim_{PINI} \hat{S}'$ whenever $m \Downarrow \hat{S}$ and $m' \Downarrow \hat{S}'$. Here, λ is the empty sequence.*

The definition of progress-sensitive noninterference, PSNI for short, differs from that of PINI only in the interpretation of equivalent output traces.

DEFINITION 4 *Two output traces $\hat{S}_1$ and $\hat{S}_2$ are progress-sensitive equivalent, denoted $\hat{S}_1 \sim_{PSNI} \hat{S}_2$, iff*

$$\hat{S}_1 =_L \hat{S}_2. \quad (8)$$

The underpinning fact in PSNI is that the attacker is assumed to be able to observe the progress status of the program.

DEFINITION 5 *Program M satisfies progress-sensitive noninterference if for any two configurations $m = (M^c, \sigma, \check{S}, \lambda)$ and $m' = (M^c, \sigma', \check{S}', \lambda)$, $\sigma =_L \sigma'$ implies $\hat{S} \sim_{PSNI} \hat{S}'$ whenever $m \Downarrow \hat{S}$ and $m' \Downarrow \hat{S}'$.*

5. Program Rewriting for PINI and PSNI

We propose rewriting algorithms to enforce PINI and PSNI in WL programs. In doing so, we first outline how such algorithms may build on program dependence graphs (PDGs). Then, we elaborate on rewriting for the two formulations of noninterference.

5.1 *Rewriting Using PDGs*

As a building block of any mechanism for enforcing noninterference, we need a machinery to identify possible information flows from high inputs to low outputs. PDGs provide such

a device where a program is represented by a directed graph in which nodes are program statements or expressions and edges denote control or data dependences between nodes. The PDG of a program reflects all dependencies among the statements of that program, but the converse is not necessarily true in the sense there may be some false positives.

The PDG is chiefly derived from the control flow graph (CFG) [1]. The CFG represents the sequence in which statements are executed. It is a directed graph with nodes representing program statements and edges representing control flows among the nodes. There are also two particular nodes Start and Stop in the CFG as the entry and exit points of the program. By identifying control and data dependences, the CFG is converted to the PDG.

A data dependence edge from X to Y in the PD is denoted by $X \overset{d}{\hookrightarrow} Y$. Similarly, a control dependence edge is noted $X \overset{c}{\hookrightarrow} Y$. An edge $X \overset{d}{\hookrightarrow} Y$ in the PDG means that Y involves a variable that has been assigned in X. Likewise, $X \overset{c}{\hookrightarrow} Y$ means that the execution of Y is controlled by the value computed at X. When the type of the edge is not of concern, we use $X \hookrightarrow Y$ without any label. As with the CFG, there exists a particular node Start in the PDG that represents the entry point to the program. A *path* from X to Y in the PDG is denoted by $X \rightsquigarrow Y$. Each path indicates a data or control dependence depending on the type of the last edge of the path. The path constructed by adding the edge $X \hookrightarrow Y$ to the path $Y \rightsquigarrow Z$ is shown by $X \hookrightarrow Y \rightsquigarrow Z$. A path $X \rightsquigarrow Y$ in the PDG indicates that there may exist a flow from X to Y.

DEFINITION 6 *Given two nodes X and Y on the PDG of program M, we say that there is a flow from X to Y if the value computed at Y or the mere execution of Y depends on the value computed at X.*

To formalize the intuitive notion of dependence in Definition 6, assume that $\mathcal{N}$ is the set of all nodes on the PDG G of program M. Moreover, let $\mathcal{Q}$ be the set of nodes on paths $Start \rightsquigarrow X$ and $\mathcal{R} \subseteq \mathcal{N} \backslash \mathcal{Q}$ be the set of nodes on paths of the form $V \rightsquigarrow W$ where $V \in \mathcal{Q}$ and W is a node with no outgoing edge in G. Then, there is a flow from X to Y if there exist two executions of M with different values at X and different values or execution statuses—reflecting whether or not the statement is executed—at Y in which the value computed at any node $Z \in \mathcal{N} \backslash (\mathcal{Q} \cup \mathcal{R})$, as well as the execution status of Z, is the same in both executions. Notice that there is a path from X to Y in the PDG if there is a flow from X to Y. The converse, however, is not necessarily true.

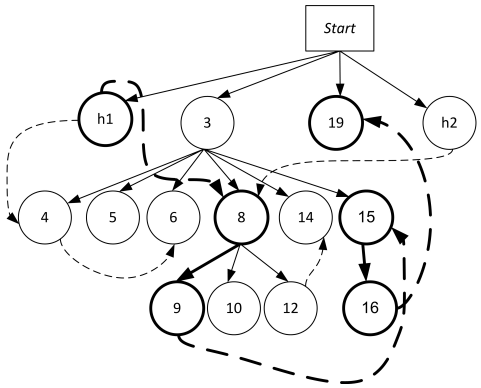
We distinguish two types of flow from X to Y concerning the path $X \rightsquigarrow Y$, *explicit* and *implicit*. An explicit flow arises if the value computed at X is directly transferred to that in Y. This may simply occur as a result of a chain of assignments on the path. An implicit flow occurs when the value computed at Y depends on whether a specific statement on the path $X \rightsquigarrow Y$ has been executed or not and the execution of that statement is controlled by the value computed at X.

DEFINITION 7 *A path $X \rightsquigarrow Y$ on the PDG of a program is said to indicate an explicit flow from X to Y if all its edges are of type data dependence. Otherwise, it is said to denote an implicit flow.*

Figure 6 shows a program and part of its PDG where control and data dependence edges are shown by solid and dashed arcs, respectively. When we use PDGs to enforce noninterference, the data dependences originated from low inputs are not of concern. Thus, the graph in Figure 6 does not contain edges reflecting such dependences. Boldface

1. $in_H\ h_1, h_2;$
2. $in_L\ l_1, l_2, l_3;$
3. $if\ \ (l_1 \leq 0)\ \ then$
4. $\quad l_2 = h_1;$
5. $\quad out_L\ l_1;$
6. $\quad out_L\ l_2;$
7. $else$
8. $\quad if(h_2 == h_1)\ then$
9. $\quad\ l_3 = 0;$
10. $\quad\ out_L\ l_1;$
11. $\quad else$
12. $\quad\ l_1 = 1;$
13. $\quad endif;$
14. $out_L\ l_1;$
15. $\quad if(l_3 == 0)\ then$
16. $\quad\ l_3 = 1;$
17. $\quad endif;$
18. $endif;$
19. $\quad out_L\ l_3;$

(a)



(b)

Figure 6. (a) A program. (b) A part of its PDG, where data dependencies originated from low variables are omitted. Boldface arcs make a path from h_1 to $out_L\ l_3$.

arcs in this figure show the path

$$h_1 \rightsquigarrow out_L\ l_3 = h_1 \overset{d}{\hookrightarrow} 8 \overset{c}{\hookrightarrow} 9 \overset{d}{\hookrightarrow} 15 \overset{c}{\hookrightarrow} 16 \overset{d}{\hookrightarrow} 19.$$

This path indicates that the value computed at Line 8 of the program depends on the value of h_1 and the execution of Line 9 depends on the value computed at 8, i.e., the Boolean value of '$h_2 == h_1$'. Our representation of PDGs is similar to that of [30].

Now, one may define a function *affect* which takes an expression or statement of a program—or equivalently, its corresponding node on the programs's PDG—and returns those expressions and statements that depend on the given expression or statement. In other words, given a node X on the PDG, the function *affect* returns a set containing all the nodes Y to which there is a path from X. A general rewriting algorithm for noninterference can then be suggested using this function, as shown in Algorithm 1. It is worth noting that the interpretation of a low observable event differs from one formulation of noninterference to the other.

A PDG, as a static representation of dependences in a program, warns us of possible illegal flows. Such flows, however, may not occur in all runs of the program. Thus, while implementing the Algorithm 1, we should regard the run-time conditions indicating whether or not a potential illegal flow addressed by PDG actually occurs at run-time. Furthermore, the implementation of the algorithm relies on our interpretation of low observable events. For example, one may consider the termination of a program as an event that can be observed by low users. In such a case, the program

$$if\ (h == l)\ then\ while\ (true)\ do\ Nop;\ done\ else\ Nop; endif$$

leaks high information and is not secure. It may also be assumed that low variables cannot be observed before the termination of the program. These are things that may

influence the way we refine the algorithm shown in Algorithm 1.

Algorithm 1: A general rewriting algorithm for noninterference.

```
1 foreach statement X producing a high input event h_in do
2 |   foreach statement Y producing a low observable event e_L do
3 |   |   if Y ∈ affect(X) then
4 |   |   |   transform Y into Y' such that Y' ∉ affect(X) in the new program.
5 |   |   end
6 |   end
7 end
```

It is worth noting that dependence graphs give us the opportunity to rely on strong guarantees and other significant characteristics backed up by ongoing research. Although WL is a simple language, dependence graphs for real-world programming languages with higher-order structures are available [24]. In particular, system dependence graphs have been proposed as a substitute for PDGs when inter-procedural analysis is required [29]. Moreover, the idea of tracking sensitive paths can be extended to complex language structures using path conditions. This means that the rewriting algorithms proposed in this paper can, in principle, be extended to full-blown languages.

5.2 *Rewriting for PINI*

We elaborate the general rewriting algorithm in Algorithm 1 so that its output program satisfies the policy defined in Definition 3. The main idea is the replacement of any out_L command with $out_L\ \bot$ or Nop provided it is affected by high inputs. As an example, $out_L\ l_2$ in the program of Figure 2 is replaced by $out_L\ \bot$ because it is influenced by $in_H\ h_1$. This change prevents leakage in the form of an explicit flow from line 2 to line 7. Such a modification, however, disregards run-time information and may be more than required. Since programs have access to run-time information, it is plausible to incorporate such information into the rewritten program. In fact, $out_L\ \bot$ should be executed instead of $out_L\ l_1$ only if $l_1 == 0$ holds.

To resolve this, we suggest the use of a variant of *path conditions* [49]. A path condition $p(X, Y)$, in our setting, is defined over program variables and gives conditions under which the flow represented by $X \rightsquigarrow Y$ actually occurs. That is, our path conditions must be true for the flow represented by the path to occur. The path conditions proposed earlier can be used to check that a path is indeed traversed at run-time. This is useful for identifying explicit flows. In case of implicit flows, however, the flow may occur at run-time even if the path is not traversed completely. This occurs when a node on the path with an incoming control dependence edge does not execute due to the value of the controlling expression. Thus, the execution of all nodes on the path indicating an implicit flow is not necessary for the flow to occur. In brief, the following holds for paths from high inputs to low outputs.

Observation 1 *The flow indicated by the path $h \rightsquigarrow out_L\ l$ on the PDG of a WL program occurs only if every node on the path with incoming data dependence edge is executed.*

That is, there are no pair of program executions delineated under Definition 6 in which some nodes with incoming data dependence edge are not executed. The implication of this observation is that all intermediate nodes on the path indicating an explicit flow should be tracked at run-time. An intermediate node on the path indicating an implicit flow, however, should be tracked only if its incoming edge is of type data dependence. As will be seen shortly, our rewriters make changes to the given program so that it can

be checked that all intermediate nodes with incoming data dependence edges on the path terminating at $out_L\ l$ commands are executed at run-time. If so, $out_L\ \bot$ or Nop is executed instead of the command. Otherwise, $out_L\ l$ itself is executed.

WL programs involve simple path conditions [30] which are derived from the execution conditions of the nodes. The execution condition for a node X is generally obtained by backtracking from X to $Start$ through control dependence edges on the path. It is a Boolean expression which is true iff X executes. The path condition $p(X, Y)$ for $X \rightsquigarrow Y$ is then defined to be the conjunction of the execution conditions of the nodes on the path. The execution and path conditions of this paper differ slightly from what explained above. In constructing execution conditions, the controlling expressions containing high variables are considered to be always true. Path conditions are also defined as the conjunction of the nodes on the path with an incoming data dependence edge. If there is no such a node, the path condition is considered to be true.

To employ path conditions more efficiently, we assume that programs only have static single assignments [**?**]. That is, they do not contain multiple assignments for a single variable. In this way, it is guaranteed that the execution condition of a node remains the same in the rest of the path. There are different algorithms for converting a given program to the one having only static single assignments. We assume that such a conversion has been applied to the inputs of our rewriting algorithms. It is also assumed that there is no loop-carried data dependences [30] in paths from high inputs to low outputs of the program. Such dependences make path conditions more complicated since different instances of the same variable as well as the number of loop iterations should appear in the conditions.

Now, we propose RW_{PINI} as a rewriter for PINI. As shown in Algorithm 2, it takes the code of a program M and its corresponding dependence graph G and returns a program code M' that satisfies progress-insensitive noninterference. The application of RW_{PINI} to the program in Figure 2 yields the one in Figure 7. Since the illegal flow from h_1 to $out_L\ l_2$ occurs only when the path condition $p(2, 7) = (l_1 == 0)$ is true, the rewritten program decides between $out_L\ l_1$ and $out_L\ \bot$ according to the value of l_1. The program in Figure 7 satisfies PINI because its output traces are progress-insensitive equivalent for any two executions of the program with the same low inputs.

Algorithm 2: RW_{PINI}: A rewriter for progress-insensitive noninterference which takes program M and its PDG G.

initialize $\mathcal{F}$ to the set of all paths $Start \hookrightarrow P \rightsquigarrow P'$ in the PDG G of M where P is the node representing a high input and P' is the node representing $out_L\ l$ for some l;
if $\mathcal{F} = \emptyset$ **then**
  return M;
end
create a copy of M, name it M', and change it as follows:
determine the type of flow indicated by each path $f \in \mathcal{F}$;
foreach $f \in \mathcal{F}$ **do**
  Generate the path condition of f as the conjunction of the execution conditions of nodes N satisfying $f = Start \rightsquigarrow X \overset{d}{\hookrightarrow} N \rightsquigarrow P'$ if there are such nodes on the path and true otherwise;
end
foreach *node* n *on* G *representing* $out_L\ l$ *for some* l **do**
  let c be the disjunction of the path conditions of all $f' \in \mathcal{F}$ which terminate at n;
  if *all paths* $f' \in \mathcal{F}$ *terminating at* n *indicate an explicit flow* **then**
    replace $out_L\ l$ with the statement "if c then $out_L\ \bot$ else $out_L\ l$ endif";
  else
    replace $out_L\ l$ with the statement "if c then Nop else $out_L\ l$ endif";
  end
end
return M'

1. $in_L\ l_1, l_2$;
2. $in_H\ h_1$;
3. $if\ (l_1 == 0)\ then$
4. $\quad l_2 = h_1$;
5. $else\ Nop$;
6. $endif$
7. $if\ (l_1 == 0)\ then\ out_L\ \bot$;
8. $else\ out_L\ l_2$;
9. $endif$

(a)

$\langle \check{0}_L, \check{0}_L, \check{0}_H, \hat{\bot}_L \rangle$
$\langle \check{0}_L, \check{0}_L, \check{1}_H, \hat{\bot}_L \rangle$
$\langle \check{0}_L, \check{1}_L, \check{0}_H, \hat{\bot}_L \rangle$
$\langle \check{0}_L, \check{1}_L, \check{1}_H, \hat{\bot}_L \rangle$
$\langle \check{1}_L, \check{0}_L, \check{0}_H, \hat{0}_L \rangle$
$\langle \check{1}_L, \check{0}_L, \check{1}_H, \hat{0}_L \rangle$
$\langle \check{1}_L, \check{1}_L, \check{0}_H, \hat{1}_L \rangle$
$\langle \check{1}_L, \check{1}_L, \check{1}_H, \hat{1}_L \rangle$

(b)

Figure 7. Rewriting the program in Figure 2 using RW_{PINI}. (a) The transformed program M'. (b) Traces of M'.

5.3 *Rewriting for PSNI*

The progress-sensitive formulation of noninterference imposes more restrictions on low observable behavior than the progress-insensitive one. Consider the program shown in Figure 8. The algorithm RW_{PINI} replaces $out_L\ l_1$ in Line 4 with Nop. The resulting program satisfies PINI. It is, however, insecure in terms of PSNI because the loop in the program may diverge depending on the value of the high input h_1. In other words, a low observer can infer the value of h_1 by observing the progress status of the program.

Hence, to enforce PSNI, it should additionally be ensured that the progress status of the program does not reveal any high information. That is, the program must always terminate or must always diverge when it begins from low-equivalent initial states. There are a number of techniques and tools to determine whether or not a program, from some specific classes of programs, terminates [15, 16, 50]. Nevertheless, the problem is in general undecidable. This is perhaps the main reason why the few solutions proposed for PSNI are too conservative and reject any program in which there is a *high-dependent loop*—a loop that the execution of its body, or the number of rounds it executes, depends on high values [40, 48, 57]. Moore et. al. [39] propose a type system together with a run-time mechanism, called termination oracle, to detect those loops whose progress status depends only on low values. Such an oracle may provide a higher precision in comparison with static solutions but at the cost of an extra run-time overhead. On the other hand, the execution of the program gets stuck if the oracle cannot predict the progress status of the loop.

We propose a rewriter that transforms programs so that the progress status of the result does not depend on high values. In this way, a program that leaks high values through its progress status is given the chance to execute, albeit its semantics may be

1. $in_H\ h_1$;
2. $in_L\ l_1$;
3. $if\ (h_1 == l_1)\ then$
4. $\quad while\ (true)\ do\ out_L\ l_1$;
   $done$
5. $else\ Nop$;
6. $endif$
7. $out_L\ l_1$;

Figure 8. A PINI-violating program. The rewriter RW_{PINI} replaces the out_L command in Line 4 with Nop but the result does not satisfy PSNI.

```
1.  in_H h1;
2.  in_L l1;
3.  while (h1 < 1) do
4.     Nop;
5.     h1 = h1 + 1;
6.  done
7.  while (true) do
8.     out_L l1;
9.     h1 = h1 + l1;
10. done
11. out_L l1;
```

(a)

```
1.  in_H h1;
2.  in_L l1;
3.  while (h1 < l1) do
4.     Nop;
5.     h1 = h1 − l1;
6.  done
7.  out_L l1;
```

(b)

Figure 9. (a) A program with a loop (Line 3) that always terminates and a loop (Line 7) that diverges in all states. (b) A loop (Line 3) that terminates in states where $l_1 < 0$ or $h_1 \geq l_1$.

changed for the sake of soundness. In WL, *while* is the only construct being responsible for divergence. Thus, we need a device—a function—to analyze *while* loops. In devising our rewriter, we assume that there is a loop analyzer which is able to statically examine the code of a given loop. The rewriting algorithm guarantees that the resulting program terminates, or diverges, for any two low-equivalent initial states.

The loop analyzer is assumed to take the code of a loop and return a Boolean expression which is true for the states in which the execution of that loop definitely terminates. It returns the constant 'True' if the loop always terminates and 'False' if it diverges in all states. For example, it returns True for the first loop in the program in Figure 9.(a) and False for the second one. Likewise, the loop analyzer returns the expression '$h_1 \geq l_1 \ \vee \ l_1 < 0$' for the loop of the program in Figure 9.(b) meaning that it may diverge in a state with $l_1 > 0$, for example. Notice that our rewriter for PSNI relies on the existence of a powerful loop analyzer. Such a device analyzes most loops successfully. Its performance also improves by new achievements in identifying the patterns indicating specific termination behaviors. Nonetheless, there may be loops for which the loop analyzer cannot return any expression. We assume that the input to the rewriter does not contain such loops.

Our rewriter for PSNI transforms the code using what is returned by the loop analyzer as well as the paths from high variables to loop guards in the PDG of the code. The rewriter leaves the loop intact if the loop analyzer returns True for that loop. The same holds for an always diverging loop, i.e., the one for which the loop analyzer returns False, provided there is no control dependence path from high inputs to the guard of that loop—recall that a control dependence path is the one whose last edge is of type control dependence. In fact, an always diverging loop may divulge high information if it is controlled by an expression which depends on high inputs. If so, the loop is replaced with an if-then construct with the same guard and body as the loop. If the loop analyzer returns an expression which is not True nor False, the rewriter conditions the execution of the loop on the expression returned. In this way, the new code definitely terminates.

Algorithm 3: RW_{PSNI}: A rewriter for progress-sensitive noninterference which takes program M and its PDG G.

initialize $\mathcal{D}$ to the set of all paths $Start \hookrightarrow P \hookrightarrow E^+$ in G where E^+ is a path terminating at a loop guard and P is the node representing a high input;
$M' = RW_{PINI}(M, G)$;
if $\mathcal{D} = \emptyset$ **then**
  return M'
end
$H = max\{height(n) \mid n \text{ is a node on } G\}$, where *height* is a function that returns the height of a given node on the tree obtained by removing data dependence edges from G;
change M' as follows:
for h=H *to* 1 **do**
  foreach *node* n *with* $height(n) = h$ *representing a loop on some path* $f \in \mathcal{D}$ **do**
    r=*LoopAnalyzer*(*loop*(n));
    if $r = False$ **then**
      if $X \overset{c}{\hookrightarrow} n$ *appears on at least one path* $f \in \mathcal{D}$ **then**
        replace *loop*(n) with the statement "if *guard*(n) then *body*(n) endif";
      end
    else
      if $r \neq True$ **then**
        replace *loop*(n) with the statement "if r then *loop*(n) endif";
      end
    end
  end
  $h = h - 1$;
end
return M'

RW_{PSNI} as shown in Algorithm 3 takes the code of program M and its corresponding dependence graph G. It returns the program code M' that satisfies PSNI. This algorithm also makes use of 'LoopAnalyzer' which is a loop analyzer as specified above. RW_{PSNI} first calls RW_{PINI}. The result of applying RW_{PINI} to M is a program that satisfies PSNI if M does not contain high-dependent loops. Otherwise, the loops may be rewritten by collecting all the paths of the form $Start \hookrightarrow h \hookrightarrow E^+$ where h is a high input and E^+ is a path terminating at a loop guard. Notice that such paths may also contain intermediate nodes representing some other loop guards. The functions $guard(n)$, $body(n)$, and $loop(n)$ return the Boolean guard, the body, and the entire loop represented by node n on the PDG, respectively.

As seen, RW_{PSNI} may transform a high-dependent loop into a conditional statement where the body of the loop executes at most once. Other strategies, such as changing the guard so that the loop body can only execute finitely many times, are also possible. Such strategies may be compared to ours via a transparency analysis. It should also be noted that nested loops are analyzed first, since the influence of their rewritten version on the termination behavior of outer loops may differ from that of the ones before rewriting. To achieve this, RW_{PSNI} uses that height of the nodes representing loops in the tree obtained by removing data dependence edges from the PDG. Figure 10 shows the result of applying RW_{PSNI} to the program shown in Figure 9.b.

6. Soundness and Transparency

Assume that M, M', G, and G' are a given program, its rewritten version, and their PDGs, respectively. Moreover, $\mathcal{F}$ and $\mathcal{F}'$ are the sets of paths of the form $Start \hookrightarrow h \rightsquigarrow out_L\ l$ in G and G' where h is a node representing a high input and l is a low variable. It is also assumed that $\mathcal{D}$ and $\mathcal{D}'$ are sets of paths of the form $Start \hookrightarrow h \hookrightarrow E^+$ in G and G' in which h is a node representing a high input and E^+ is a path terminating at

1. $in_H\ h_1$;
2. $in_L\ l_1$;
3. $if(l_1 \leq h_1\ \vee\ l_1 < 0)\ then$
4. $\quad while\ (h_1 < l_1)\ do$
5. $\quad\quad Nop$;
6. $\quad\quad h_1 = h_1 - l_1$;
7. $\quad done$
8. $endif$
9. $out_L\ l_1$;

Figure 10. Program of Figure 9.b rewritten by RW_{PSNI}.

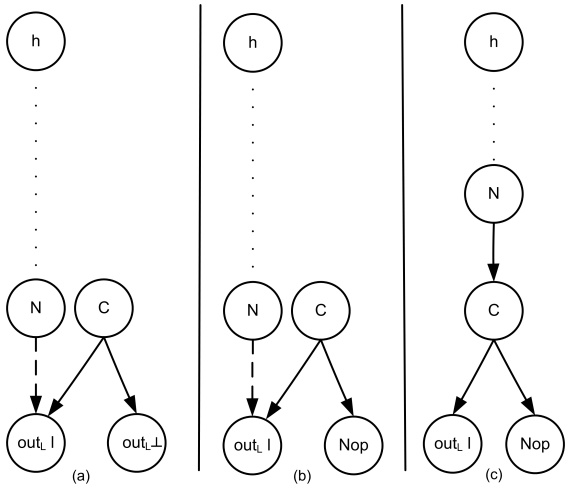


Figure 11. Possible configurations for a path from a high input to an out_L command in the PDG G' of $RW_{PINI}(M, G)$. The nodes C and N represent the controlling node inserted by the rewriter and the last node before the $out_L\ l$ command on a path in G, respectively. (a) Every path in G terminating at $out_L\ l$ indicates an explicit flow. (b) Some path in G terminating at $out_L\ l$ represents an implicit flow and $out_L\ l$ is data dependent to N. (c) Some path in G terminating at $out_L\ l$ represents an implicit flow and $out_L\ l$ is control dependent to N.

a loop guard. We begin by the proof of soundness for PINI.

PROPOSITION 6.1 *The flow indicated by the path $f' \in \mathcal{F}'$ in the PDG of $RW_{PINI}(M, G)$ does not occur.*

Proof. Let $n(f')$ be the set of nodes on f' whose incoming edge is of type data dependence. From Observation 1, the flow indicated by f' occurs only if

$$\bigwedge_{X \in n(f')} e(X), \tag{9}$$

where $e(X)$ is the execution condition of X. Figure 11 shows possible configurations of $f' \in \mathcal{F}'$ in the PDG G' of $RW_{PINI}(M, G)$ in which N is the last node on f' before $out_L\ l$. The node C also represents the path condition of f' which is added by the PINI rewriter such that it controls the execution of $out_L\ l$. If C does not hold, the condition (9) is not satisfied, and therefore, the flow does not occur. If C holds, (9) is not satisfied in Cases (a) and (b) because the execution condition for $out_L\ l$, whose incoming edge is of type data dependence in these cases, is not satisfied. The only remaining case is when C holds and the incoming edge of $out_L\ l$ is of type control dependence. The following

is the reasoning for this case. As C holds Nop is executed instead of $out_L\ l$. We claim that in any other execution with the same low values and possibly a different value of h the command Nop is executed instead of $out_L l$, and therefore, the flow does not occur. This is immediate as path conditions are indeed defined on low variables. That is, an execution condition is regarded as true when the corresponding execution is conditioned on a high value. Thus, changing a high value while low values remain unchanged does not make C false. ■

THEOREM 1 *Let M be a program and G its PDG. Then, $M' = RW_{PINI}(M, G)$ satisfies progress-insensitive noninterference.*

Proof. Assume that the initial states σ_1 and σ_2 in two initial configurations m_1 and m_2 are low-equivalent, $m_1 \Downarrow \hat{S}_1$, $m_2 \Downarrow \hat{S}_2$ and $\hat{S}_1 \approx_{PINI} \hat{S}_2$. Then, from (7), we have

$$\exists i > 0.\Big((\hat{S}_1)_L[..i-1] = (\hat{S}_2)_L[..i-1] \\ \wedge \ \circlearrowright \neq (\hat{S}_1)_L[i] \neq (\hat{S}_2)_L[i] \neq \circlearrowright \Big), \tag{10}$$

where $(\hat{S}_1)_L$ and $(\hat{S}_2)_L$ denote the subsequences obtained from $\hat{S}_1$ and $\hat{S}_2$ by restricting them to low events. Thus, there are computations

$$\mathcal{T}_1 : (M'^c, \sigma_1, \check{S}_1, \lambda) \Rightarrow^* (M'^r, \sigma_1', \check{S}_1, \hat{S}_1)$$

and

$$\mathcal{T}_2 : (M'^c, \sigma_2, \check{S}_2, \lambda) \Rightarrow^* (M'^s, \sigma_2', \check{S}_2, \hat{S}_2),$$

where '$\Rightarrow^*$' is the transitive closure of '$\Rightarrow$' and M'^r and M'^s denote the remaining sequences of commands of M'^c when the two computations have led to exactly i low-observable events. Since WL is a deterministic language, $(\hat{S}_1)_L[i] \neq (\hat{S}_2)_L[i]$ and $\sigma_1 =_L \sigma_2$ imply that there is a high variable h in the input block such that $\sigma_1(h) \neq \sigma_2(h)$. Furthermore, there exists at least one $out_L\ l$ command that leads to the inequality of events $(\hat{S}_1)_L[i]$ and $(\hat{S}_2)_L[i]$. This implies that there is a paths $f' = Start \hookrightarrow h \rightsquigarrow out_L\ l$ in $\mathcal{F}'$ where $out_L\ l$ is the command whose execution leads to one of (or both) the events $(\hat{S}_1)_L[i]$ and $(\hat{S}_2)_L[i]$. Assume that $(\hat{S}_1)_L[i]$ is generated by $out_L\ l$. There are two possibilities for $(\hat{S}_2)_L[i]$. If $(\hat{S}_2)_L[i]$ is generated by the same $out_L\ l$ command, the flow indicated by f' is an explicit flow occurred at run-time. Otherwise, it is generated by a different out_L command indicating that an implicit flow has occurred at run-time. Both cases contradict Proposition 6.1. Thus, such computations do not exist and the assumption $\hat{S}_1 \approx_{PINI} \hat{S}_2$ does not hold. This completes the proof. ■

Now, we prove that RW_{PSNI} is sound. Since RW_{PSNI} first applies RW_{PINI} to the given program, the resulting rewritten code, before applying the instructions of the block beginning at line 8 of the algorithm, satisfies PINI. In addition to having PINI-equivalent output traces, PSNI also requires every pair of runs from low-equivalent initial states to have the same progress status, termination or silent divergence. The proof is based on the properties of the PDG of the output program as well as the semantics of WL programs.

THEOREM 2 *Let M be a program with statically decidable loop termination behaviors and G its PDG. Then, $M' = RW_{PSNI}(M, G)$ satisfies progress-sensitive noninterference.*

Proof. Assume that the initial states σ_1 and σ_2 of two initial configurations m_1 and m_2 are low-equivalent, $m_1 \Downarrow \hat{S}_1$, $m_2 \Downarrow \hat{S}_2$, and $\hat{S}_1 \neq_L \hat{S}_2$. Then,

$$\exists i > 0.\Big((\hat{S}_1)_L[..i-1] = (\hat{S}_2)_L[..i-1] \ \wedge \ (\hat{S}_1)_L[i] \neq (\hat{S}_2)_L[i]\Big), \tag{11}$$

where $(\hat{S}_1)_L$ and $(\hat{S}_2)_L$ denote the subsequences obtained from $\hat{S}_1$ and $\hat{S}_2$ by restricting them to low events. RW_{PSNI} invokes RW_{PINI} and does not create any new path of the form $Start \hookrightarrow h \rightsquigarrow out_L \ l$ for some high input h and low variable l. Therefore, from Theorem 1, it is concluded that paths in G' from high variables to out_L commands cannot result in $(\hat{S}_1)_L[i] \neq (\hat{S}_2)_L[i]$. Thus,

$$\circlearrowleft = (\hat{S}_1)_L[i] \ \oplus \ (\hat{S}_2)_L[i] = \circlearrowleft, \tag{12}$$

where $\oplus$ denotes exclusive or. Without loss of generality, we assume that $(\hat{S}_1)_L[i] = \circlearrowleft$. Now, consider the computations $\mathcal{T}_1$ and $\mathcal{T}_2$ of M' producing i low events such that their first $i-1$ low events are the same and their ith low events are $(\hat{S}_1)_L[i] = \circlearrowleft$ and $(\hat{S}_2)_L[i] \neq \circlearrowleft$, respectively. Since WL is deterministic and $\sigma_1 =_L \sigma_2$, there is at least one high input h such that $\sigma_1(h) \neq \sigma_2(h)$. Furthermore, there are paths $f' \in \mathcal{D}'$ of the form $Start \hookrightarrow h \hookrightarrow E^+$ such that E^+ contains a node representing a loop that produces $\circlearrowleft$ in $\mathcal{T}_1$. This means that the termination behavior of the loop depends on h. Thus, it is either an always diverging loop controlled by some node on paths $f' \in \mathcal{D}'$ or a loop that diverges for a subset of input variables. None of these cases is possible as RW_{PSNI} replaces such loops with terminating "if-then" statements. Thus, $\mathcal{T}_1$ does not exist and M' satisfies PSNI. ■

To analyze the transparency of our rewriters, we first remind that a path from a high variable to a low output in the PDG may not denote an actual dependency. This is due to the very nature of static analysis methods and is not limited to PDGs. For example, assume that the command $out_L \ l_1$ is conditioned on a very hard decision problem involving some high inputs. We may then act in a conservative manner and add the corresponding path to the PDG, though the event raised by $out_L \ l_1$ may not depend on the high inputs. Fortunately, PDG construction tools are supported by advanced code optimization techniques [24] eliminating many of such imprecisions. Notwithstanding, in our analysis of transparency, we merely focus on the underpinning logic of the proposed algorithms. Thus, we evaluate transparency of our rewriters under the assumption that input WL programs are those for which PDG construction tools yield perfect PDGs. Such a PDG perfectly reflects dependencies in the program and contains the path $X \rightsquigarrow Y$ if and only if there is a flow from X to Y.

A program rewriter is defined to be transparent if it preserves good features of programs. To capture this notion, we define an appropriate preorder relation on programs according to the given security policy. Transparency then stipulates that the rewritten program must be higher than or equal to the input program on the preorder relation. As stated in Section 3, the preorder relation on programs is defined in terms of the abstract values returned by an abstraction function $\mathcal{A}$. For a given program represented by its set of traces ψ, $\mathcal{A}(\psi)$ is intended to reflect the desired characteristics of ψ.

Definition 8 *The abstraction function for a formulation NI of noninterference is defined to be the function $\mathcal{A}_{NI} : 2^{\Psi} \rightarrow 2^{\Psi}$ which returns the set of those traces of the given program that, in terms of NI, are compatible with any trace of that program. That*

is,

$$\mathcal{A}_{NI}(\psi) = \left\{ S \in \psi \mid \forall S' \in \psi.\ \check{S} =_L \check{S}' \Rightarrow \hat{S} \sim_{NI} \hat{S}' \right\}. \tag{13}$$

The preorder relation $\sqsubseteq_{NI}$ on programs for a formulation NI of noninterference is then defined by

$$\psi \sqsubseteq_{NI} \psi' \Leftrightarrow \mathcal{A}_{NI}(\psi) \subseteq \mathcal{A}_{NI}(\psi'). \tag{14}$$

DEFINITION 9 *A rewriter RW is said to correctively enforce the formulation NI of noninterference for the set of programs $\Delta \subseteq 2^{\Psi}$ if for any program $\psi \in \Delta$, $RW(\psi)$ satisfies NI and $\psi \sqsubseteq_{NI} RW(\psi)$.*

It is worth noting that our rewriters for PINI and PSNI take a program together with its PDG, while a rewriter is defined to be a total function taking a sole program. To resolve this difference, one may think of our rewrites as the ones that first derive the given PDG from the given program and then transform the program by using that PDG. Also note that we use programs and their trace sets interchangeably. Thus, in the following, $RW_{NI}(M, G)$ has the same meaning as $RW_{NI}(\psi)$ where ψ is the trace set of M and NI is $PINI$ or $PSNI$. $RW_{NI}(\psi)$ can be interpreted as the rewritten program or its traces as well.

DEFINITION 10 *The PDG of a program is said to be perfect if its any path $X \rightsquigarrow Y$ implies the existence of a flow from X to Y and vice versa.*

THEOREM 3 *RW_{PINI} correctively enforces PINI for WL programs with perfect PDGs.*

pt1. Let ψ be the trace set of program M and ψ' be that of $RW_{PINI}(M, G)$ where G is the perfect PDG of M. Notice that as RW_{PINI} is sound, we have $\mathcal{A}_{PINI}(\psi') = \psi'$. Now, assume that M satisfies PINI. This implies that $\mathcal{A}_{PINI}(\psi) = \psi$ and, as G is the perfect PDG of M, there is no path of the form $Start \hookrightarrow h \rightsquigarrow out_L\ l$ in G. Therefore, $RW_{PINI}(M, G) = M$ and $\psi' = \psi$. Hence, $\mathcal{A}_{PINI}(\psi) \subseteq \mathcal{A}_{PINI}(\psi')$ and, in turn, $\psi \sqsubseteq_{PINI} RW_{PINI}(\psi)$. Another case is when M does not satisfy PINI. Assume that $S \in \mathcal{A}_{PINI}(\psi)$. This means that there is no trace in ψ that is incompatible, in terms of PINI, with S. Thus, there is no output event in S produced by an out_L command for which there is a path of the form $Start \hookrightarrow h \rightsquigarrow out_L\ l$ in G. Hence, the events in S remain intact in ψ' since RW_{PINI} only changes those output commands to which there are paths from high inputs. ■

Now, we prove that RW_{PSNI} correctively enforces PSNI. In the following theorem, we consider programs whose loops can be successfully analyzed by LoopAnalyzer. That is, the loop analyzer in RW_{PSNI} returns a Boolean expression for any loop in those programs.

THEOREM 4 *RW_{PSNI} correctively enforces PSNI for WL programs with perfect PDGs whose loops, if any, can be successfully analyzed by LoopAnalyzer.*

pt2. The proof is similar to that of Theorem 3 except that paths of the form $Start \hookrightarrow h \hookrightarrow E^+$ terminating at loop guards also appear in the argument. ■

Notice that more accurate abstraction functions can be devised to constrain the manner in which the rewriter is allowed to change a bad trace. For example, the abstraction

function may also return the traces obtained by removing low outputs from bad traces. An appropriate preorder relation may then prohibit the modification of events other than low outputs. In fact, while low outputs in a bad trace can be removed or replaced with $\perp$, the other events should remain intact.

7. Conclusion

We demonstrate that the information-flow policies appropriate for programs with observable intermediate values can be enforced through rewriting. To do so, we devise rewriting algorithms for progress-insensitive and progress-sensitive noninterference. We prove that the proposed rewriters are sound and transparent in the paradigm of corrective security policy enforcement where valid aspects of programs are preserved. Indeed, this paper takes a first step towards extending the notion of transparent policy enforcement to so-called hyperproperties. However, there is still much to be done. Extending the ideas presented in this paper to the languages supporting classes, objects, method invocation, multithreading, and other features of modern languages deserves future research. Characterizing the policies enforceable by rewriting in a corrective enforcement paradigm is another challenging problem.

References

[1] F.E. Allen, *Control flow analysis*, in *ACM Sigplan Notices*, Vol. 5, 1970, pp. 1–19.
[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, *Termination-insensitive noninterference leaks more than just a bit*, in *Computer Security - ESORICS 2008*, Lecture Notes in Computer Science, Vol. 5283, Springer-Verlag Berlin, Heidelberg, 2008, pp. 333–348.
[3] T.H. Austin and C. Flanagan, *Efficient purely-dynamic information flow analysis*, in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, ACM, 2009, pp. 113–124.
[4] T.H. Austin and C. Flanagan, *Permissive dynamic information flow analysis*, in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* no. 3 PLAS '10, ACM, 2010, pp. 1–12.
[5] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu, *Enforceable security policies revisited*, ACM Transactions on Information and System Security (TISSEC) 16 (2013), pp. 1–26.
[6] D. Beauquier, J. Cohen, and R. Lanotte, *Security policies enforcement using finite edit automata*, Electronic Notes in Theoretical Computer Science (ENTCS) 229 (2009), pp. 19–35.
[7] D. Beauquier, J. Cohen, and R. Lanotte, *Security policies enforcement using finite and pushdown edit automata*, International Journal of Information Security 12 (2013), pp. 1–18.
[8] L. Bello and E. Bonelli, *On-the-Fly Inlining of Dynamic Dependency Monitors for Secure Information Flow*, in *Formal Aspects in Security and Trust*, Lecture Notes in Computer Science, Vol. 7140, Springer-Verlag Berlin, Heidelberg, 2011, pp. 55–69.
[9] L. Beringer, *End-to-end multilevel hybrid information flow control*, in *Programming Languages and Systems*, Springer, 2012, pp. 50–65.
[10] N. Bielova and F. Massacci, *Predictability of enforcement*, in *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ESSoS'11, Springer-Verlag, 2011, pp. 73–86.
[11] A. Bohannon, B.C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, *Reactive noninterference*, in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, ACM, 2009, pp. 79–90.
[12] A. Chudnov and D.A. Naumann, *Information Flow Monitor Inlining*, in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, IEEE, 2010, pp. 200–214.
[13] M.R. Clarkson and F.B. Schneider, *Hyperproperties*, Journal of Computer Security - 7th International Workshop on Issues in the Theory of Security (WITS'07) 18 (2010), pp. 1157–1210.

[14] E.S. Cohen, *Strong dependency : a formalism for describing information transmission in computational systems*, Tech. Rep., Computer Science Department, Carnegie Mellon University, 1976.
[15] B. Cook, A. Podelski, and A. Rybalchenko, *Termination proofs for systems code*, SIGPLAN Not. 41 (2006), pp. 415–426.
[16] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv, *Proving Conditional Termination*, in *Computer Aided Verification*, Lecture Notes in Computer Science, Vol. 5123, Springer-Verlag Berlin, Heidelberg, 2008, pp. 328–340.
[17] B. De Sutter, B. De Bus, and K. De Bosschere, *Link-time binary rewriting techniques for program compaction*, ACM Transactions on Programming Languages and Systems 27 (2005), pp. 882–945.
[18] G.S. Dennis Volpano Cynthia Irvine, *A sound type system for secure flow analysis*, Journal of Computer Security 4 (1996), pp. 167–187.
[19] D. Devriese and F. Piessens, *Noninterference through Secure Multi-execution*, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, IEEE, 2010, pp. 109–124.
[20] U. Erlingsson, *The inlined reference monitor approach to security policy enforcement*, Ph.D. thesis, Cornell University Ithaca, NY, USA, 2004.
[21] U. Erlingsson and F.B. Schneider, *SASI enforcement of security policies: A retrospective*, in *Proceedings of the 1999 workshop on New security paradigms*, 1999, pp. 87–95.
[22] J. Ferrante, K.J. Ottenstein, and J.D. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programing Languages and Systems 9 (1987), pp. 319–349.
[23] J.A. Goguen and J. Meseguer, *Security policies and security models*, in *Proceedings of IEEE Symposium on Security and Privacy*, Vol. 12, IEEE, 1982, pp. 11–18.
[24] C. Hammer and G. Snelting, *Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs*, International Journal of Information Security 8 (2009), pp. 399–422.
[25] S. Hussein, P. Meredith, and G. Roşlu, *Security-policy monitoring and enforcement with javamop*, in *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, ACM, 2012, pp. 3–4.
[26] G.M. Kevin W. Hamlen and F.B. Schneider, *Computability classess for enforcement mechanisms*, ACM Transactions on Programming Languages and Systems 28 (2006), pp. 175–205.
[27] R. Khoury and N. Tawbi, *Corrective enforcement: A new paradigm of security policy enforcement by monitors*, ACM Transactions on Information and System Security 15 (2012), pp. 1–27.
[28] R. Khoury and N. Tawbi, *Which security policies are enforceable by runtime monitors? a survey*, Computer Science Review 6 (2012), pp. 27–45.
[29] J. Krinke, *Advanced slicing of sequentioal and concurrent programs*, Ph.D. thesis, University of Passau, 2003.
[30] J. Krinke, *Advanced Slicing of Sequential and Concurrent Programs*, in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, IEEE, 2004, pp. 464–468.
[31] G. Le Guernic, A. Banerjee, T. Jensen, and D.A. Schmidt, *Automata-based confidentiality monitoring*, in *Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*, ASIAN'06, Vol. 4435, Springer-Verlag Berlin, Heidelberg, 2007, pp. 75–89.
[32] J. Ligatti, L. Bauer, and D. Walker, *Edit automata: Enforcement mechanisms for run-time security policies*, International Journal of Information Security 4 (2005), pp. 2–16.
[33] J. Ligatti, L. Bauer, and D. Walker, *Run-time enforcement of nonsafety policies*, ACM Transactions on Information and System Security (TISSEC) 12 (2009), pp. 1–41.
[34] J. Ligatti and S. Reddy, *A theory of runtime enforcement, with results*, in *ESORICS'10 Proceedings of the 15th European conference on Research in computer security*, Springer-Verlag Berlin, Heidelberg, 2010, pp. 87–100.
[35] J. Magazinius, A. Russo, and A. Sabelfeld, *On-the-fly inlining of dynamic security monitors*, Computers and Security 31 (2012), pp. 827 – 843.
[36] Y. Mallios, L. Bauer, D. Kaynar, and J. Ligatti, *Enforcing More with Less: Formalizing Target-Aware Run-Time Monitors*, in *Security and Trust Management*, Lecture Notes in Computer Science, Vol. 7783, Springer-Verlag Berlin, Heidelberg, 2013, pp. 17–32.
[37] H. Mantel and H. Sudbrock, *Types vs. pdgs in information flow analysis*, in *Logic-Based Program Synthesis and Transformation*, Springer, 2013, pp. 106–121.
[38] D. McCullough, *Specifications for multi-level security and a hook-up*, in *IEEE Symposium on Security and Privacy*, IEEE, 1987, pp. 161–166.
[39] S. Moore, A. Askarov, and S. Chong, *Precise enforcement of progress-sensitive security*, in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM,

2012, pp. 881–893.

[40] K. O'Neill, M. Clarkson, and S. Chong, *Information-flow security for interactive programs*, in *Computer Security Foundations Workshop, 2006. 19th IEEE*, CSFW '06, IEEE, 2006, pp. 190–201.

[41] F. Rafailidis, I. Panagos, A. Arvanitidis, and P. Katsaros, *Inlined monitors for security policy enforcement in web applications*, in *Proceedings of the 17th Panhellenic Conference on Informatics*, ACM, 2013, pp. 75–82.

[42] A. Russo and A. Sabelfeld, *Dynamic vs. Static Flow-Sensitive Security Analysis*, in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, IEEE, 2010, pp. 186–199.

[43] A. Sabelfeld and A.C. Myers, *Language-based information-flow security*, IEEE Journal of Selected Areas in Communications 21 (2006), pp. 5–19.

[44] J.F. Santos and T. Rezk, *An information flow monitor-inlining compiler for securing a core of javascript*, in *ICT Systems Security and Privacy Protection*, Springer, 2014, pp. 278–292.

[45] F.B. Schneider, *Enforceable security policies*, ACM Transactions on Information and System Security (TISSEC) 3 (2000), pp. 30–50.

[46] F.B. Schneider, J.G. Morrisett, and R. Harper, *A Language-Based Approach to Security*, in *Informatics - 10 Years Back. 10 Years Ahead*, Springer-Verlag Berlin, Heidelberg, 2001, pp. 86–101.

[47] P. Shroff, S. Smith, and M. Thober, *Dynamic Dependency Monitoring to Secure Information Flow*, in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, IEEE, 2007, pp. 203–217.

[48] G. Smith and D. Volpano, *Secure information flow in a multi-threaded imperative language*, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, ACM, 1998, pp. 355–364.

[49] G. Snelting, T. Robschink, and J. Krinke, *Efficient path conditions in dependence graphs for software safety analysis*, ACM Transactions on Software Engineering and Methodoly 15 (2006), pp. 410–457.

[50] F. Spoto, F. Mesnard, and E. Payet, *A termination analyzer for java bytecode based on path-length*, ACM Transactions on Programming Languages and Systems 32 (2010), pp. 1–70.

[51] M. Sridhar, R. Wartell, K.W. Hamlen, S.M. Khan, K.W. Hamlen, K.W. Hamlen, L. Kagal, M. Kantarcioglu, R. Wartell, K.W. Hamlen, *et al.*, *Hippocratic binary instrumentation: First do no harm*, in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Vol. 1, 2012, pp. 126–140.

[52] 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 *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, 2004, pp. 243–254.

[53] V.N. Venkatakrishnan, W. Xu, D.C. DuVarney, and R. Sekar, *Provably correct runtime enforcement of non-interference properties*, in *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS'06, Springer-Verlag Berlin, Heidelberg, 2006, pp. 332–351.

[54] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham, *Efficient software-based fault isolation*, in *ACM SIGOPS Operating Systems Review*, Vol. 27, 1994, pp. 203–216.

[55] D. Wasserrab, D. Lohner, and G. Snelting, *On PDG-based noninterference and its modular proof*, in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, ACM, 2009, pp. 31–44.

[56] S. Zdancewic and A.C. Myers, *Observational determinism for concurrent program security*, in *Proceedings of IEEE Computer Security Foundations Workshop*, IEEE, 2003, pp. 29–43.

[57] D. Zhang, A. Askarov, and A.C. Myers, *Predictive mitigation of timing channels in interactive systems*, in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, 2011, pp. 563–574.