Journal of Universal Computer Science, vol. 22, no. 7 (2016), 956-991
submitted: 8/3/15, accepted: 30/6/16, appeared: 1/7/16 © *J.UCS*

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

Afshin Lamei
(Department of Computer Engineering and Information Technology
Amirkabir University of Technology (Tehran Polytechnic)
P.O.Box: 15875-4413, Tehran, Iran
lamei@aut.ac.ir)

Mehran S. Fallah
(Department of Computer Engineering and Information Technology
Amirkabir University of Technology (Tehran Polytechnic)
P.O.Box: 15875-4413, Tehran, Iran
msfallah@aut.ac.ir)

Abstract: Program rewriting is defined as transforming a given program into one satisfying some intended properties. This technique has recently been suggested as a means for enforcing security policies. In this paper, we propose rewriting mechanisms based on program dependence graphs to enforce noninterference in programs with observable intermediate values. We first formulate progress-insensitive and progress-sensitive noninterference for the programs of a model language. Then, we give rewriting mechanisms that correctively enforce such policies. The notion of corrective enforcement is also introduced. It is indeed a realization of transparent rewriting in which the good behaviors of the program are preserved irrespective of whether the program is secure or not. Unlike purely static mechanisms, our rewriting mechanisms allow tracking those points on dependence graphs that are actually traversed at run-time, thereby achieving transparency. The rewriting-based enforcement of noninterference also obviates the need for changing the run-time system, something that cannot be avoided in dynamic enforcement mechanisms. The proposed rewriters are provably sound and transparent for the class of programs whose loops can be analyzed for termination and any dependency in their dependence graphs definitely reflects the existence of a flow.

Key Words: Corrective enforcement; noninterference; program dependence graphs; program rewriting.
Category: F.3.1, D.4.6, F.4

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 [Sutter et al. 2005]. Recently, it has been suggested as an effective method for enforcing *security policies* [Schneider et al. 2001]. 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 family of sets of executions, that is, a set of programs, where an execution is an arbitrary sequence of states. In other words, security policies are *hyperproperties* [Clarkson and Schneider 2010]. Some security policies, such as access control policies, are *properties* in the sense that they can be characterized as a single 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. There are, however, important information flow policies that cannot be expressed as properties. The Goguen-Meseguer noninterference [Goguen and Meseguer 1982], generalized noninterference [McCullough 1987], and observational determinism [Zdancewic and Myers 2003], which are collectively known as *noninterference* policies, are a few examples.

This paper concentrates on the enforcement of noninterference policies. Such policies basically state that a low observer (an attacker), who knows the program and can only observe public run-time events, can learn nothing about high (private) inputs to the program. A noninterference policy indeed demands that the run-time observations of the attacker should be equivalent in every pair of program executions that agree on public inputs. The definition of low-equivalent observations is where things become interesting and leads to different notions of noninterference.

Many static and dynamic mechanisms have been proposed to enforce noninterference policies. Among the static mechanisms, security type systems [Dennis Volpano 1996, Hunt and Sands 2006, Van Delft et al. 2015] have been studied more than the others. They are sound but overly conservative in the sense that they reject many healthy programs. Although such mechanisms do not impose any run-time overhead, they usually require complicated type annotations. Dynamic mechanisms, such as execution monitoring [Le Guernic et al. 2007, Shroff et al. 2007, Austin and Flanagan 2010] and secure multi-execution [Devriese and Piessens 2010], have access to run-time information, and thus, are more permissive than static ones. Such mechanisms, however, inflict execution overhead and may require substantial changes in the run-time environment; secure multi-execution, for example, requires programs to be run in environment with special schedulers. There are also various hybrid mechanisms which make use of static information at run-time for the sake of more precision [Venkatakrishnan et al. 2006, Beringer 2012, Buirs et al. 2015].

Another approach to the enforcement of noninterference is program rewriting. The idea of secure multi-execution has been incorporated into a static transformation technique based on self-composition [Barthe et al. 2012]. There are other attempts, e.g., [Bello and Bonelli 2011, Magazinius et al. 2012, Chudnov and Naumann 2010, Chudnov and Naumann 2015], at in-lining information flow monitors into programs. In this paper, we devise novel program rewriting mechanisms for the enforcement of noninterference in programs with observable intermediate values. The program rewriters we propose make use of program dependence graphs (PDGs) [Ferrante et al. 1987, Krinke 2003], which are proven at least as powerful as security type systems in detecting potential information flows [Hammer and Snelting 2009, Mantel and Sudbrock 2013]. It has been proven that the expression represented by node X on a PDG has no influence on that represented by node Y if there is no path from X to Y [Wasserrab et al. 2009].

Unlike monitor in-lining, PDG-based rewriting does not introduce shadow variables, program counters, and boxes into the code. It only makes use of path conditions for some specific nodes on the PDG. Moreover, static information like dependencies among statements helps us track information leakage via program termination. Monitors, however, can barely detect such information leaks. The rewriters proposed in this paper do not require the original program to be run multiple times. This is in contrast to secure multi-execution which requires multiple executions of the program, one for each security level. In fact, it is conceivable that PDG-based rewriting incurs less performance overhead than secure multi-execution if there is a large number of security levels and a small number of paths from higher-level variables to lower-level outputs.

The rewriter based on secure multi-execution [Barthe et al. 2012] produces one single program, but it may change the order of output commands of the original program which are not at the same security level. It also embeds a specific scheduler into the code. In contrast to black-box enforcement techniques like secure multi-execution, PDG-based rewriting can change any command rather than just manipulating I/O.

The policies investigated in this paper are *progress-insensitive* and *progress-sensitive* noninterference [Askarov et al. 2008]. They 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 meaning 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 research on information-flow security, e.g., [Russo and Sabelfeld 2010, Dennis Volpano 1996, Sabelfeld and Myers 2006], focuses mainly on variants of progress-insensitive noninterference. There are also few solutions for progress-sensitive noninterference [O'Neill et al. 2006, Smith and Volpano 1998, Zhang et al. 2011, Bohannon et al. 2009, Devriese and Piessens 2010] most of which are excessively restrictive static mechanisms that reject many healthy programs only due to the existence of control-flow dependencies on high values.

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. Our rewriters are provably sound and transparent for the class of programs with perfect PDGs (as defined in Definition 10) whose loops can be analyzed for termination—it is shown that our rewriters are sound and transparent up to the precision of the methods and tools, already devised in the literature, for deriving program dependence graphs and for analyzing the termination behavior of loops. A rewriter is sound if its output certainly satisfies the given policy. To deal with transparency more effectively, we introduce the concept of *corrective enforcement* for our formulations of noninterference—it is indeed an extension of the same concept for security properties [Khoury and Tawbi 2012a]. 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. This is in contrast to earlier rewriters proposed for enforcing noninterference which do not care about the extent to which a rewriter may change an insecure program.

In brief, this paper has the following contributions.

- Novel rewriting mechanisms based on program dependence graphs are proposed to enforce progress-insensitive and progress-sensitive noninterference. This is a first attempt at incorporating PDGs into rewriting-based enforcement of noninterference policies, to the best of our knowledge.
- A new paradigm of security policy enforcement through rewriting is formalized. According to this notion of enforcement, transparency is redefined in such a way that the set of possible executions of the transformed program should be as close as possible to that of the input program whether the input program is secure or not.
- The proposed rewriters are provably sound and transparent for the programs with perfect PDGs whose loops can be analyzed for termination.
- An implementation of the proposed rewriting algorithms is also provided [WLR 2016].

We proceed as follows: Section [2 Related Work] is an overview of the related work. Section [3 Preliminaries] gives basic definitions and some motivating examples of rewriting as a security mechanism. Section [4 Security] gives a formalization of progress-insensitive and progress-sensitive noninterference in the model language introduced in the same section. Section [5 Program Rewriting for PINI and PSNI] 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 Security]. Section [6 Soundness and Transparency] is on the soundness and transparency of our rewriting mechanisms. Section [7 Conclusion] concludes the paper.

2 Related Work

A great deal of research, starting with [Hammer 2009], has been dedicated to the use of PDGs in verifying and enforcing information flow policies. Indeed, PDGs are proven as powerful as security type systems in detecting potential illegal flows [Hammer and Snelting 2009, Mantel and Sudbrock 2013]. As with other static mechanisms, the use of PDGs involves false positives. To mitigate such an imprecision in practice, scholars have proposed the application of so-called path conditions [Snelting et al. 2006]. By using path conditions, it can be ascertained that if the information flows implied by PDGs actually occur at run-time. However, path conditions also suffer from false positives. Taghdiri et al. [Taghdiri et al. 2011] propose the use of SAT solvers to attain more precise path conditions. The idea of generating more accurate path conditions is also applicable to PDG-based rewriting. However, their work differs from ours in the sense that it uses PDGs and path conditions to verify programs and not to transform them to secure ones. In fact, some of the techniques they propose to reduce false positives, such as multiple execution of the input program and limiting the analysis and refinement of path conditions to a user-provided timeout, cannot be used in program rewriting.

Program dependence graphs have also been used for information flow control in high-level programming languages [Hammer and Snelting 2009, Johnson et al. 2015]. Nevertheless, the use of PDGs in program rewriting in such a way that the resulting programs satisfy a given noninterference policy is proposed and implemented for the first time in the current paper.

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 [Vachharajani et al. 2004] is an assembly-level rewriting mechanism to track implicit and explicit flows at run-time. Beringer [Beringer 2012] 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.

A hybrid transformation method for the enforcement of noninterference has been proposed in [Venkatakrishnan et al. 2006]. 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. A framework has also been devised for in-lining security monitors while the program is being executed [Magazinius et al. 2012]. 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 a code transformer to be available at run-time so that an appropriate monitor can be in-lined in the code which is dynamically generated.

Secure multi-execution (SME) [Devriese and Piessens 2010] is a dynamic technique for the enforcement of time- and termination-sensitive noninterference. It runs multiple copies of the given program simultaneously, one per each security level. The security guarantee of SME relies on an appropriate scheduling strategy to control the concurrent execution of the program copies [Kashyap et al. 2011]. The authors give two noninterference policies: normal or time-insensitive which only regards terminating runs of the program and strong or time-sensitive which can deal with timing channels. Excluding external timing channels which are realized by measuring the time through a watch, the strong noninterference is similar to our progress-sensitive noninterference in the sense that both policies can deal with internal timing channels as well as the leakage of sensitive information raised by silent divergence. A SME-based program transformation method [Barthe et al. 2012] has also been proposed that eliminates the need for modifying the run-time system, something required by SME.

Our PDG-based rewriting mechanisms differ from the SME-based transformation in several respects. The mechanisms proposed in the current paper do not introduce local copies of variables nor do buffer input commands. Unlike the SME-based rewriting, our rewriters do not change the order of outputs at different security levels [Zanarini et al. 2013]. Moreover, SME does not necessarily achieve its claimed security guarantees when there exist noncomparable security levels [Kashyap et al. 2011]. PDG-based rewriting, on the contrary, provides security even in the case of noncomparable security levels. The SME-based rewriter also embeds a scheduler into the code which is not trivial. Nevertheless, one good feature of SME, compared to PDG-based rewriting, is that there are no constraints on the programs it can be applied to.

It has been proven that there is no purely dynamic mechanism to enforce flow-sensitive noninterference [Russo and Sabelfeld 2010]—the concept of noninterference adapted for systems in which security classes can be updated dynamically. 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 [Bello and Bonelli 2011] suggest in-lining a dynamic dependency monitor [Shroff et al. 2007] into the code. In this way, they can go beyond policies such as *no-sensitive* [Austin and Flanagan 2009] and *permissive* [Austin and Flanagan 2010] *upgrade*. To detect an illegal flow, however, their proposal may require several runs of the program, something that is not possible in many applications. Santos and Rezk [Santos and Rezk 2014] implement the no-sensitive upgrade policy for termination-insensitive noninterference.

Le Guernic et al. [Le Guernic et al. 2007] 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. such a mechanism has been implemented by Chudnov and Naumann [Chudnov and Naumann 2010] who propose a hybrid monitor for flow-sensitive noninterference 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. [Russo and Sabelfeld 2010]. This may lead to an unacceptable run-time overhead. The mechanism does not inhibit termination channels either. A comparison of the proposed rewriters with hybrid monitors in terms of transparency is given in Section 6.2.

A taxonomy of information flow monitors including those we addressed above has been provided by Bielova and Rezk [Bielova and Rezk 2016]. They compare the security guarantees of different dynamic mechanisms to enforce information flow policies. The authors also formalize the notion of Termination Aware Noninterference (TANI) which distinguishes between the termination channel present in the original program and the one that is added by the monitor. To compare different mechanisms, they define true and false transparency. The false transparency cares about the way a monitor produces outputs in response to insecure executions. This concept is captured by the notion of corrective enforcement we propose in the current paper.

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, $\sigma_0 \ldots \sigma_i$, 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 set $\mathcal{P}$ of all policies 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 a policy $P \in \mathcal{P}$, 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 [Hamlen et al. 2006]. The concept of transparency defined by (3) lacks an important feature that seems essential to program rewriters. It puts no restriction on the way a rewriter is allowed to transform the programs not complying with the policy. We will further discuss the notion of transparency by introducing the paradigm of corrective enforcement of security policies through rewriting in Section [6 Soundness and Transparency].

Now, we give an example of how a noninterference policy, which is usually not a property, may be enforced through a program rewriting mechanism based on PDGs. Fig. 1.a shows a program written in WL, the language we will introduce later in this paper. In this program, L and H represent low and high values, respectively. The program violates noninterference since it contains two illegal flows from the high input h_1 to the low outputs produced by the commands $out_L\ l_1$ and $out_L\ l_2$. In fact, the commands at Lines 4 and 11 depend on the command at Line 2. Such dependencies are reflected by the PDG of the program on which there are paths from the node representing Line 2 to nodes representing Lines 4 and 11. The dependencies reflected by the two paths, however, are of different natures, control and data. The program rewriter uses PDGs to infer the kind of dependency, and then, rewrites the program accordingly.

```
1.  in_L l1, l2;
2.  in_H h1;
3.  if (h1 == 0) then
4.      out_L l1
5.  else Nop
6.  endif;
7.  if (l1 == 0) then
8.      l2 = h1
9.  else Nop
10. endif;
11. out_L l2
```

(a)

```
1.  in_L l1, l2;
2.  in_H h1;
3.  if (h1 == 0) then
4.      Nop
5.  else Nop
6.  endif;
7.  if (l1 == 0) then
8.      l2 = h1
9.  else Nop
10. endif;
11. if (l1 == 0) then out_L ⊥
12. else out_L l2
13. endif
```

(b)

Figure 1: An insecure program and its rewritten, secure version.

The program of Fig.1.b is the output of the rewriters of this paper when their input is the program of Fig.1.a. As seen, the rewriters replace the low output command at Line 4 with *Nop* representing no operation. The low output command at Line 11, however, is replaced with the conditional structure of Lines 11 and 12 in the rewritten program. Indeed, the rewritten program outputs a default value $\perp$ whenever there is an explicit flow from h_1 to l_2 and outputs the same value as the original program when there is not such a flow. It is evident that there is no pair of traces of the rewritten program with the same low inputs and different low outputs. That is, the following noninterference policy is satisfied by the set ψ_M of the traces of the program of Fig.1.b, where $t_{l_{in}}$ and $t_{l_{out}}$ are the sequences of low inputs and low outputs generated by trace t, respectively.

$$\forall t, t' \in \psi_M .\ t_{l_{in}} = t'_{l_{in}} \ \Rightarrow\ t_{l_{out}} = t'_{l_{out}}. \tag{4}$$

As seen, 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. 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.

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 we formalize our notions through a model language, dependence graphs for real-world programming languages with higher-order structures are available [Hammer and Snelting 2009]. In particular, system dependence graphs have been proposed as a substitute for PDGs when inter-procedural analysis is required [Krinke 2003]. 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 more advanced programming languages.

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. 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 Fig. 2. 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_Γ *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 '$out_\Gamma\ \perp$' that outputs the constant $\perp$ differing from all other constants of the language. The most important control structure of WL is *while*, hence the name.

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 $\circlearrowright$, in case it is observable to low users. The set of all possible sequences of events a program M may generate is denoted by $\mathcal{S}(M)$. Fig. 3 is an example program and all sequences of events it may generate as well as part of its PDG where it is assumed that variables only take 0 or 1 and data dependencies originated from low variables are omitted. 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. Each node of Fig. 3.c containing a

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

Figure 2: Syntax of WL.

number corresponds to the statement at the same line of the program. There is also a special node for each high variable which helps us to track dependencies on that variable. Control and data dependencies are shown by solid and dashed arcs, respectively.

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 Fig. 3 does not satisfy noninterference 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.

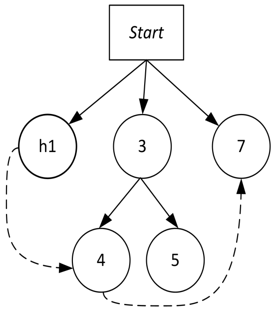
Formal semantics of WL bears on how a program produces a trace of events at run-time. As shown in Fig. 4, it builds on configurations where a configuration is either a terminal configuration $(\epsilon, \sigma, \check{S}, \hat{S})$ or an intermediate one $(c, \sigma, \check{S}, \hat{S})$ where ϵ denotes 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 '$\longrightarrow$'.

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)

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

(b)



(c)

Figure 3: A program, its traces, and a part of its PDG. (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 $\Gamma \in \{L, H\}$. (c) A part of its PDG, where data dependencies originated from low variables are omitted.

Let X be the set of variables. A state σ is defined to be a mapping $\sigma : X \rightarrow \mathbb{N} \cup \{\text{null}\}$, where $\mathbb{N}$ is the set of all integers and '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 [Askarov et al. 2008]. 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 Fig. 5.

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

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

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

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

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

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

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

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

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

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

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

Figure 4: Small-step semantics for WL.

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

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

Figure 5: Rules defining the relation '⇒'.

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 Fig. 6. As programs may produce infinitely many observable events, the rules are interpreted *coinductively*.

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

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

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

Figure 6: 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*

$$\big(\hat{S}_1 =_L \hat{S}_2\big) \vee \exists \hat{S}, \hat{S}', \hat{S}''.\ \Big(\big(\hat{S} =_L \hat{S}'\big) \wedge \Big(\big(\hat{S}_1 = \hat{S} \cdot \langle \circlearrowleft \rangle \wedge \hat{S}_2 = \hat{S}' \cdot \hat{S}''\big) \vee \big(\hat{S}_2 = \hat{S} \cdot \langle \circlearrowleft \rangle \wedge \hat{S}_1 = \hat{S}' \cdot \hat{S}''\big)\Big)\Big). \quad (5)$$

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.*

It can easily be shown that Definition 3 and the termination-insensitive noninterference defined in [Askarov et al. 2008] are equivalent—such an equivalence has indeed been stated through the third item of Proposition 1 in the same paper. 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*

$$\hat{S}_1 =_L \hat{S}_2. \tag{6}$$

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) [Allen 1970]. 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 PDG 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.

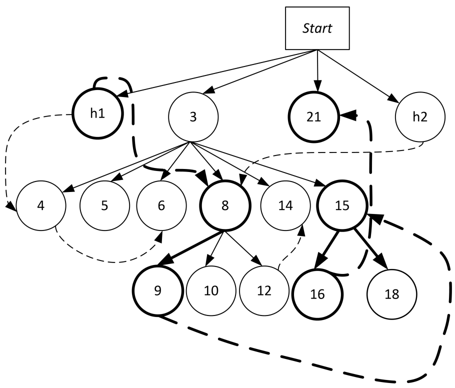
Definition 6 *Given two nodes X and Y on the PDG of program M, we say that there is a flow from X to Y if the value computed at Y or the mere execution of Y depends on the value computed at X.*

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 $Start \rightsquigarrow X$ except for $Start$. Assume also that $\mathcal{R}$ is the set of nodes other than $Start$ in $\mathcal{N} \backslash \mathcal{Q}$ which are on paths of the form $V \rightsquigarrow W$ where $V \in \mathcal{Q}$ and W is a node with no outgoing edge in G. Evidently, $\mathcal{R} \subseteq \mathcal{N} \backslash \mathcal{Q}$ and the set $\mathcal{N} \backslash (\mathcal{Q} \cup \mathcal{R})$ comprises the nodes of G that are not branched from the nodes on paths from $Start$ to X. 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.

It is worth noting that there is a flow from X to Y only if there is a path from X to Y. This result was formally proven in [Wasserrab et al. 2009]. The converse, however, is not necessarily true. That is, the existence of paths from X to Y may not reflect the existence of a flow from X to Y. PDG-based rewriting converts a program into one whose PDG either contains no paths from high variables to low outputs or contains such paths but they do not reflect any flow.

```
in_H h1, h2;
in_L l1, l2, l3;
if (l1 <= 0) then
    l2 = h1;
    out_L l1;
    out_L l2
else
   if(h2 == h1) then
     l3 = 0;
     out_L l1
   else
     l1 = 1
   endif;
   out_L l1;
   if(l3 == 0) then
     l3 = 1
   else
     Nop
   endif;
endif;
out_L l3
```

(a)



(b)

Figure 7: (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 $out_L\ l_3$.

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.*

Fig. 7 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

Algorithm 1: A general rewriting algorithm for noninterference.

foreach *statement* X *producing a high input event* h_{in} **do**
    foreach *statement* Y *producing a low observable event* e_L **do**
        if $Y \in \mathit{affect}(X)$ **then**
            transform Y into Y' such that $Y' \notin \mathit{affect}(X)$ in the new program.
        end
    end
end

not of concern. Thus, the graph in Fig. 7 does not contain edges reflecting such dependences. Boldface arcs in this figure show the path

$$h_1 \rightsquigarrow out_L\ l_3 = h_1 \overset{d}{\hookrightarrow} 8 \overset{c}{\hookrightarrow} 9 \overset{d}{\hookrightarrow} 15 \overset{c}{\hookrightarrow} 16 \overset{d}{\hookrightarrow} 21.$$

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 [Krinke 2004].

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

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

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.

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 $out_L \perp$ or *Nop* provided it is affected by high inputs. As an example, $out_L\ l_2$ in the program of Fig. 3 is replaced by $out_L \perp$ 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, $out_L \perp$ 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* [Snelting et al. 2006]. 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, $out_L \perp$ or *Nop* is executed instead of the command. Otherwise, $out_L\ l$ itself is executed.

WL programs involve simple path conditions [Krinke 2004] 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.

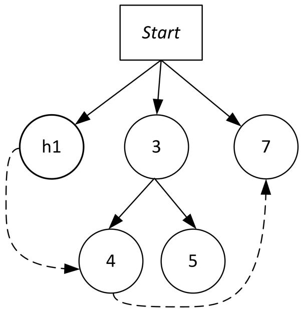
Algorithm 2: RW_{PINI}: A rewriter for progress-insensitive noninterference which takes program M and its PDG G.

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 $out_L\ \perp$ 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'

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—we may also use $p(m)$ as the path condition of the path m. The execution and path conditions of this paper differ slightly from what explained above. In constructing execution conditions, the Boolean expressions containing high variables are considered to be always true. Path conditions are also defined as the conjunction of the execution conditions 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 [Cytron et al. 1991, Taghdiri et al. 2011]. 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. The two algorithms are also assumed to only take inputs that are free of loop-carried data dependences. Section 5.5 discusses how one may extend the algorithms to handle such data dependences.

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 Fig. 3 yields the one in Fig. 8. Since the illegal flow from h_1 to $out_L\ l_2$ occurs only when the path condition



(a)

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. $endif$

(b)

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

(c)

Figure 8: Rewriting the program in Fig. 3 using RW_{PINI}. (a) Part of the PDG of the input program where dependencies originated from low variables are omitted. (b) The transformed program M'. (c) Traces of M'.

$p(2,7) = (l_1 == 0)$ 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 Fig. 8 satisfies PINI because its output traces are progress-insensitive equivalent for any two executions of the program with the same low inputs.

Note that a high-dependent output command is replaced by $out_L\ \bot$ only if all the paths to it indicate explicit flows. Otherwise, the presence or absence of $\hat{\bot}_L$, which is generated by a path indicating an implicit flow, violates PINI. If there is a path from high variables to a low output command which indicates an implicit flow, the output command will be replaced with *Nop*. It is obvious that one can replace the out_L commands which depend on high values with *Nop* regardless of the type of the flow. That is, a constant output $\bot$ may be interpreted as equivalent to *Nop* in the view of the attacker in case of explicit flows. However, similar to some previous works such as [Le Guernic et al. 2007], we choose to preserve the mere existence of such output commands whenever they does not result in violation of soundness. In particular, an interpretation of transparency may require the same number of output events at run-time for an insecure execution and its corresponding execution of the rewritten program.

Note that SME [Devriese and Piessens 2010] runs (in parallel) two copies of the program of Fig. 3, one for the high and one for the low security level. The copies for the high and low security levels are the same as the program of Fig. 3 except that Line 7 is removed for the high level and Line 2 is replaced with $in_H\ \bot$ for the low level—we have abused the syntax of WL and used $in_H\ \bot$ instead of $v_{default}$ in SME. It can easily be verified that the set of traces of the (parallel) execution of the two copies is the same as that given in Fig. 8.c.

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

Figure 9: 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.

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 Fig. 9. 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 [Cook et al. 2006, Spoto et al. 2010, Cook et al. 2008]. 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 [O'Neill et al. 2006, Smith and Volpano 1998, Zhang et al. 2011]. Moore et. al. [Moore et al. 2012] 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 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

```
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 10: (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 $h_1 \geq l_1$.

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 Fig. 10.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 Fig. 10.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

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

```
initialize 𝒟 to the set of all paths Start ↪ P ↪ 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 𝒟 = ∅ then
    return M′
end
H = max{height(n) | n 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 ∈ 𝒟 do
        r=LoopAnalyzer(loop(n));
        if r = False then
            if X ↪(c) n appears on at least one path f ∈ 𝒟 then
                replace loop(n) with the statement "if guard(n) then body(n) endif";
            end
        else
            if r ≠ True then
                replace loop(n) with the statement "if r then loop(n) endif";
            end
        end
    end
    h = h − 1;
end
return M′
```

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.

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

$in_H\ h_1;$
$in_L\ l_1;$
$if(l_1 \leq h_1\ \vee\ l_1 < 0)\ then$
$\quad while\ (h_1 < l_1)\ do$
$\quad\quad Nop;$
$\quad\quad h_1 = h_1 - l_1$
$\quad done$
$endif;$
$out_L\ l_1$

Figure 11: Program of Fig. 10.b rewritten by RW_{PSNI}.

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. Fig. 11 shows the result of applying RW_{PSNI} to the program shown in Fig. 10.b.

5.4 Multilevel Security

Algorithms 2 and 3 scale to multilevel security with small changes. We explain how Algorithm 2 can be modified to support more than two security levels making an arbitrary lattice. The same argument applies to Algorithm 3. Let $(\Gamma, \rhd)$ be the finite lattice of security levels. An information flow is allowed from γ to γ' only if $\gamma \rhd \gamma'$. The complement of $\rhd$ can be thought of as the noninterference relation in the sense that there should be no information flow from inputs at security level γ to the outputs at security level γ' iff $(\gamma, \gamma') \notin \rhd$. For a security level γ, we also define H_γ as the set of all security levels γ' which do not have the relation $\rhd$ with γ, that is, the security levels that are higher than or noncomparable to γ.

For any output command $out_\gamma\ x$ in a given program, the rewriter for PINI finds all paths on the PDG of the program from input variables at security levels $\gamma' \in H_\gamma$ to $out_\gamma\ x$. After determining the type of the flow corresponding to each path and computing the path conditions (the execution conditions involving variables with security levels in H_γ are considered to be always true), the PINI rewriter replaces $out_\gamma\ x$ with appropriate statements similar to that in the Algorithm 2 devised for the lattices of security levels having only the two elements L and H. If all the paths reflect explicit flows, the algorithm substitutes a conditional structure for the command $out_\gamma\ x$. The condition part is the disjunction of the path conditions, the if-part is $out_\gamma\ \bot$, and the else-part is $out_\gamma\ x$. If some paths reflect implicit flows, the if-part will be *Nop* instead of $out_\gamma\ \bot$.

5.5 Loop-carried data dependence

The proposed rewriting algorithms could scale to programs with loop-carried data dependences. Such a data dependence occurs when the body of a loop executes more than once causing the value computed at a node of the PDG to depend on the value of another node computed in previous iterations of the same loop. If we do not consider such dependencies, the path conditions may not reflect the conditions required for a flow to actually occur. A solution to this problem is to decompose path conditions at loop-carried data dependence edges [Krinke 2004].

Let $m = m_1 m_2 ... m_n$ be a path such that m_i's are paths without loop-carried data dependences and that each m_i is connected to m_{i+1} by a loop-carried data dependence edge. Then, the path condition of m would be

$$p(m) = \bigwedge_{1 \leq i \leq n} p(m_i, i), \tag{7}$$

where $p(m_i, i)$ denotes the path condition $p(m_i)$ for path m_i in which every occurrence of a variable v is replaced by its new instance v_i. In fact, v_i stores the value of the variable v at iteration i of the loop. Since the proposed rewriting algorithms produce an output program, such variable instances should be inserted inside the loop and properly be updated at run-time to hold the value of v at iterations i. In this way, path conditions are correctly evaluated.

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 a loop guard. We begin with the proof of soundness for PINI and PSNI. Next, we formalize and prove transparency for these rewriting mechanisms.

6.1 Soundness

Proposition 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

$$\bigwedge_{X \in n(f')} e(X), \tag{8}$$

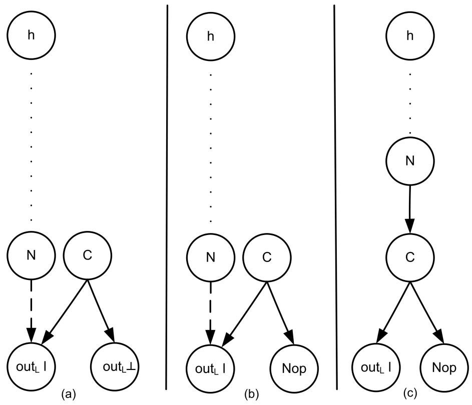


Figure 12: Possible configurations for a path from a high input to an out_L command in the PDG G' of $RW_{PINI}(M, G)$. The nodes C and N represent the controlling node inserted by the rewriter and the last node before the out_L l command on a path in G, respectively. (a) Every path in G terminating 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.

where $e(X)$ is the execution condition of X. Fig. 12 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 for a particular mapping of variable to values in an execution, the condition (8) is not satisfied for that execution, and therefore, the flow does not occur. If C holds for an execution, (8) 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 for that execution. The only remaining case is when C holds for an execution 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. $\square$

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 $\hat{S}_1 \not\approx_{PINI} \hat{S}_2$. Then, from (5), we have

$$\exists i > 0.\Big((\hat{S}_1)_L[..i-1] = (\hat{S}_2)_L[..i-1] \\ \wedge\ \circlearrowright \neq (\hat{S}_1)_L[i] \neq (\hat{S}_2)_L[i] \neq \circlearrowright \Big), \tag{9}$$

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 1. Thus, such computations do not exist and the assumption $\hat{S}_1 \not\approx_{PINI} \hat{S}_2$ does not hold. This completes the proof. $\square$

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,

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

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. 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,

$$\circlearrowright = (\hat{S}_1)_L[i] \ \oplus \ (\hat{S}_2)_L[i] = \circlearrowright, \qquad (11)$$

where $\oplus$ denotes exclusive or. Without loss of generality, we assume that $(\hat{S}_1)_L[i] = \circlearrowright$. 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 $(\hat{S}_1)_L[i] = \circlearrowright$ and $(\hat{S}_2)_L[i] \neq \circlearrowright$, 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 $\circlearrowright$ 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. $\square$

6.2 Transparency

The concept of transparency defined by (3) 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 [Bielova and Massacci 2011, Khoury and Tawbi 2012a]. 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

$$\forall \psi, \psi' \subseteq \Psi.\ \psi \sqsubseteq \psi' \Leftrightarrow \mathcal{A}(\psi) \preceq \mathcal{A}(\psi'). \tag{12}$$

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

$$\forall \psi \subseteq \Psi.\, \psi \sqsubseteq RW(\psi). \tag{13}$$

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 [Khoury and Tawbi 2012b], to program rewriting.

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 [Hammer and Snelting 2009] 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.

To define transparency, we first give 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 above, 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 $\mathcal{A}_{NI} : 2^{\Psi} \to 2^{\Psi}$ 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,*

$$\mathcal{A}_{NI}(\psi) = \left\{ S \in \psi \mid \forall S' \in \psi.\ \check{S} =_L \check{S}' \Rightarrow \hat{S} \sim_{NI} \hat{S}' \right\}. \tag{14}$$

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

$$\psi \sqsubseteq_{NI} \psi' \Leftrightarrow \mathcal{A}_{NI}(\psi) \subseteq \mathcal{A}_{NI}(\psi'). \tag{15}$$

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.*

Proof. 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. It follows 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. □

According to Theorem 3, RW_{PINI} acts transparently if the PDG of the input program reflects actual dependencies. To illustrate the point, we make use of the following programs drawn from [Bielova and Rezk 2016, Hedin et al. 2015]. Consider the following program.

$$if\ (h == l)\ then\ l = 0\ else\ l = 0\ endif;\ out_L\ l$$

It satisfies PINI since it always outputs 0. However, given an imperfect PDG, RW_{PINI} changes the output to *Nop* . Similarly, for the program

$$l = 1;\ \mathit{if}\ (h\ != \ h)\ \mathit{then}\ l = h\ \mathit{endif};\ \mathit{out}_L\ l,$$

which contains dead code, an imperfect PDG indicates a flow. Thus, RW_{PINI} converts the output command to *Nop*. Furthermore, inability to detect nontermination in situations such as the following program makes the rewriter to behave conservatively by substituting *Nop* for $out_L\ l$.

$$l = 1;\ \mathit{if}\ (h == 0)\ \mathit{then}\ \mathit{while}(\mathit{true})\ \mathit{do}\ l = h\ \mathit{done}\ \mathit{endif};\ \mathit{out}_L\ l$$

Notice that a perfect PDG of this program does not indicate a flow from $l = h$ to $out_L\ l$.

By applying the achievements of ongoing research on PDG-based information flow analysis, one may improve the transparency of RW_{PINI}. The same holds for hybrid monitors where the transparency of the monitor can be improved if we can better identify high contexts [Besson et al. 2016].

Another point is the way RW_{PINI} treats some secure programs compared to SME. Consider the following program, from [Bielova and Rezk 2016], that satisfies PINI.

$$\begin{array}{l} \mathit{if}\ (l == 1)\ \mathit{then} \\ \qquad \mathit{while}\ (h == 0)\ \mathit{do}\ Nop\ \mathit{done} \\ \mathit{else} \\ \qquad \mathit{while}\ (h == 0)\ \mathit{do}\ Nop\ \mathit{done} \\ \mathit{endif}; \\ \mathit{out}_L\ l \end{array}$$

SME modifies the executions of this program to eliminate termination channels—by the default high value $h = 1$, SME eliminates the low output 1. However, RW_{PINI} does not change the executions of this program since the PDG does not indicate any flow from h to $out_L\ l$.

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.*

Proof. 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. $\square$

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 propose program rewriters that enforce the information flow policies appropriate for programs with observable intermediate values. To do so, we devise rewriting algorithms based on program dependence graphs for progress-insensitive and progress-sensitive noninterference. We prove that our rewriters are sound and transparent for the class of programs with perfect PDGs whose loops can be analyzed for termination. To do so, we introduce and formalize the paradigm of corrective security policy enforcement as an interpretation of soundness and transparency. According to this paradigm, valid aspects of programs should be preserved. 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

[Allen 1970] Allen, F. E.: "Control flow analysis"; ACM Sigplan Notices; volume 5; 1–19; ACM, 1970.

[Askarov et al. 2008] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: "Termination-insensitive noninterference leaks more than just a bit"; Computer Security - ESORICS 2008; volume 5283 of Lecture Notes in Computer Science; 333–348; Springer-Verlag Berlin, Heidelberg, 2008.

[Austin and Flanagan 2009] Austin, T. H., Flanagan, C.: "Efficient purely-dynamic information flow analysis"; Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security; PLAS '09; 113–124; ACM, 2009.

[Austin and Flanagan 2010] Austin, T. H., Flanagan, C.: "Permissive dynamic information flow analysis"; Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security; number 3 in PLAS '10; 1–12; ACM, 2010.

[Barthe et al. 2012] Barthe, G., Crespo, J., Devriese, D., Piessens, F., Rivas, E.: "Secure multi-execution through static program transformation"; Formal Techniques for Distributed Systems;186–202; Springer, 2012.

[Bello and Bonelli 2011] Bello, L., Bonelli, E.: "On-the-fly inlining of dynamic dependency monitors for secure information flow"; Formal Aspects in Security and Trust; volume 7140 of Lecture Notes in Computer Science; 55–69; Springer-Verlag Berlin, Heidelberg, 2011.

[Beringer 2012] Beringer, L.: "End-to-end multilevel hybrid information flow control"; Programming Languages and Systems; 50–65; Springer, 2012.

[Besson et al. 2016] Besson, F., Bielova, N., Jensen, T.; "Hybrid Monitoring of Attacker Knowledge."; 29th IEEE Computer Security Foundations Symposium, 2016.

[Bielova and Massacci 2011] Bielova, N., Massacci, F.: "Predictability of enforcement"; Proceedings of the Third International Conference on Engineering Secure Software and Systems; ESSoS'11; 73–86; Springer-Verlag, 2011.

[Bielova and Rezk 2016] Bielova, N., Rezk, T.: "A Taxonomy of Information Flow Monitors"; Principles of Security and Trust ;46–67;Springer-Verlag, 2016.

[Bohannon et al. 2009] Bohannon, A., Pierce, B. C., Sjöberg, V., Weirich, S., Zdancewic, S.: "Reactive noninterference"; Proceedings of the 16th ACM Conference on Computer and Communications Security; CCS '09; 79–90; ACM, 2009.

[Buirs et al. 2015] Buiras, P., Vytiniotis, D., Russo, A.; "HLIO: Mixing static and dynamic typing for information-flow control in Haskell."; Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ACM; 2015.

[Chudnov and Naumann 2010] Chudnov, A., Naumann, D. A.: "Information flow monitor inlining"; Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium; CSF '10; 200–214; IEEE, 2010.

[Chudnov and Naumann 2015] Chudnov, A. and Naumann, D.; "Inlined Information Flow Monitoring for JavaScript"; Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security,CCS '15; ACM; 2015.

[Clarkson and Schneider 2010] Clarkson, M. R., Schneider, F. B.: "Hyperproperties"; Journal of Computer Security - 7th International Workshop on Issues in the Theory of Security (WITS'07); 18 (2010), 6, 1157–1210.

[Cook et al. 2008] Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: "Proving conditional termination"; Computer Aided Verification; volume 5123 of Lecture Notes in Computer Science; 328–340; Springer-Verlag Berlin, Heidelberg, 2008.

[Cook et al. 2006] Cook, B., Podelski, A., Rybalchenko, A.: "Termination proofs for systems code"; SIGPLAN Not.; 41 (2006), 6, 415–426.

[Cytron et al. 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K.: "Efficiently computing static single assignment form and the control dependence graph"; ACM Transactions on Programming Languages and Systems (TOPLAS); 13 (1991), 4, 451–490.

[Dennis Volpano 1996] Dennis Volpano, G. S., Cynthia Irvine: "A sound type system for secure flow analysis"; Journal of Computer Security; 4 (1996), 2, 167–187.

[Devriese and Piessens 2010] Devriese, D., Piessens, F.: "Noninterference through secure multi-execution"; Proceedings of the 2010 IEEE Symposium on Security and Privacy; SP '10; 109–124; IEEE, 2010.

[Erlingsson and Schneider 1999] Erlingsson, U., Schneider, F. B.: "Sasi enforcement of security policies: A retrospective"; Proceedings of the 1999 workshop on New security paradigms; 87–95; ACM, 1999.

[Ferrante et al. 1987] Ferrante, J., Ottenstein, K. J., Warren, J. D.: "The program dependence graph and its use in optimization"; ACM Transactions on Programing Languages and Systems; 9 (1987), 3, 319–349.

[Goguen and Meseguer 1982] Goguen, J. A., Meseguer, J.: "Security policies and security models"; Proceedings of IEEE Symposium on Security and Privacy; volume 12; 11–18; IEEE, 1982.

[Hammer 2009] Hammer, C.: "Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs"; PhD Thesis; Universitat Karlsruhe (TH), Fak. f. Informatik; 2009.

[Hammer and Snelting 2009] Hammer, C., Snelting, G.: "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs"; International Journal of Information Security; 8 (2009), 6, 399–422.

[Hamlen et al. 2006] Hamlen, K., Morrisett, G., Schneider, F. B.: "Computability classess for enforcement mechanisms"; ACM Transactions on Programming Languages and Systems; 28 (2006), 1, 175–205.

[Hedin et al. 2015] Hedin, D., Bello, L., Sabelfeld, A,;"Value-sensitive hybrid information flow control for a JavaScript-like language"; Computer Security Foundations

Symposium (CSF), 2015 IEEE 28th. 351–365; IEEE, 2015.

[Hunt and Sands 2006] Hunt, S. and Sands, D.; "On flow-sensitive security types."; ACM SIGPLAN Notices. Vol. 41. No. 1; ACM, 2006.

[Johnson et al. 2015] Johnson, A., Waye, L., Moore, S., Chong, S.: "Exploring and Enforcing Security Guarantees via Program Dependence Graphs"; Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation; 291–302; ACM, 2015.

[Kashyap et al. 2011] Kashyap, V., Wiedermann, B., Hardekopf, B.: "Timing-and termination-sensitive secure information flow: Exploring a new approach"; Security and Privacy (SP), 2011 IEEE Symposium on; 413–428; IEEE, 2011.

[Khoury and Tawbi 2012a] Khoury, R., Tawbi, N.: "Corrective enforcement: A new paradigm of security policy enforcement by monitors"; ACM Transactions on Information and System Security; 15 (2012a), 2, 1–27.

[Khoury and Tawbi 2012b] Khoury, R., Tawbi, N.: "Which security policies are enforceable by runtime monitors? a survey"; Computer Science Review; 6 (2012b), 1, 27–45.

[Krinke 2003] Krinke, J.: Advanced slicing of sequentioal and concurrent programs; Ph.D. thesis; University of Passau (2003).

[Krinke 2004] Krinke, J.: "Advanced slicing of sequential and concurrent programs"; Proceedings of the 20th IEEE International Conference on Software Maintenance; ICSM '04; 464–468; IEEE, 2004.

[Le Guernic et al. 2007] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D. A.: "Automata-based confidentiality monitoring"; Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues; volume 4435 of ASIAN'06; 75–89; Springer-Verlag Berlin, Heidelberg, 2007.

[Magazinius et al. 2012] Magazinius, J., Russo, A., Sabelfeld, A.: "On-the-fly inlining of dynamic security monitors"; Computers & Security; 31 (2012), 7, 827 – 843.

[Mantel and Sudbrock 2013] Mantel, H., Sudbrock, H.: "Types vs. pdgs in information flow analysis"; Logic-Based Program Synthesis and Transformation; 106–121; Springer, 2013.

[McCullough 1987] McCullough, D.: "Specifications for multi-level security and a hook-up"; IEEE Symposium on Security and Privacy; 161–166; IEEE, 1987.

[Moore et al. 2012] Moore, S., Askarov, A., Chong, S.: "Precise enforcement of progress-sensitive security"; Proceedings of the 2012 ACM Conference on Computer and Communications Security; CCS '12; 881–893; ACM, 2012.

[O'Neill et al. 2006] O'Neill, K., Clarkson, M., Chong, S.: "Information-flow security for interactive programs"; Computer Security Foundations Workshop, 2006. 19th IEEE; CSFW '06; 190–201; IEEE, 2006.

[Russo and Sabelfeld 2010] Russo, A., Sabelfeld, A.: "Dynamic vs. static flow-sensitive security analysis"; Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium; CSF '10; 186–199; IEEE, 2010.

[Sabelfeld and Myers 2006] Sabelfeld, A., Myers, A. C.: "Language-based information-flow security"; IEEE Journal of Selected Areas in Communications; 21 (2006), 1, 5–19.

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

[Schneider et al. 2001] Schneider, F. B., Morrisett, J. G., Harper, R.: "A language-based approach to security"; Informatics - 10 Years Back. 10 Years Ahead; 86–101; Springer-Verlag Berlin, Heidelberg, 2001.

[Shroff et al. 2007] Shroff, P., Smith, S., Thober, M.: "Dynamic dependency monitoring to secure information flow"; Proceedings of the 20th IEEE Computer Security Foundations Symposium; CSF '07; 203–217; IEEE, 2007.

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

[Snelting et al. 2006] Snelting, G., Robschink, T., Krinke, J.: "Efficient path conditions in dependence graphs for software safety analysis"; ACM Transactions on Software Engineering and Methodoly; 15 (2006), 4, 410–457.

[Spoto et al. 2010] Spoto, F., Mesnard, F., Payet, E.: "A termination analyzer for java bytecode based on path-length"; ACM Transactions on Programming Languages and Systems; 32 (2010), 8, 1–70.

[Sutter et al. 2005] Sutter, D., De Bus, B., De Bosschere, K.: "Link-time binary rewriting techniques for program compaction"; ACM Transactions on Programming Languages and Systems; 27 (2005), 5, 882–945.

[Taghdiri et al. 2011] Taghdiri, M., Snelting, G., Sinz, C.: "Information flow analysis via path condition refinement";Formal Aspects of Security and Trust; 65–79; Springer, 2011.

[Vachharajani et al. 2004] Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M., August, D. I.: "Rifle: An architectural framework for user-centric information-flow security"; Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on; 243–254; IEEE, 2004.

[Van Delft et al. 2015] van Delft, B., Hunt, S., Sands, D.; "Very Static Enforcement of Dynamic Policies."; Principles of Security and Trust. 32-52; Springer Berlin Heidelberg; 2015.

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

[Wahbe et al. 1994] Wahbe, R., Lucco, S., Anderson, T. E., Graham, S. L.: "Efficient software-based fault isolation"; ACM SIGOPS Operating Systems Review; volume 27; 203–216; ACM, 1994.

[Wasserrab et al. 2009] Wasserrab, D., Lohner, D., Snelting, G.: "On pdg-based noninterference and its modular proof"; Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security; PLAS '09; 31–44; ACM, 2009.

[WLR 2016] Formal Security Lab, Tehran Polytechnic: While Language Rewriter. (2016) `http://ceit.aut.ac.ir/formalsecurity/rewriter/tool/WLRewriter.html`.

[Zanarini et al. 2013] Zanarini, D., Jaskelioff, M., Russo, A.: "Precise enforcement of confidentiality for reactive systems";Computer Security Foundations Symposium (CSF), IEEE 26th; 18–32; IEEE, 2013.

[Zdancewic and Myers 2003] Zdancewic, S., Myers, A. C.: "Observational determinism for concurrent program security"; Proceedings of IEEE Computer Security Foundations Workshop; 29–43; IEEE, 2003.

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