Implicit flows in malicious and nonmalicious code

Alejandro RUSSO [a], Andrei SABELFELD [a] Keqin LI [b]
[a] *Chalmers University of Technology, Sweden*
[b] *SAP Research, France*

Abstract. Information-flow technology is a promising approach for ensuring security by design and construction. When tracking information flow, of particular concern are *implicit* flows, i.e., flows through control flow when computation branches on secret data and performs publicly observed side effects depending on which branch is taken.

The large body of literature exercises two extreme views on implicit flows: either track them (striving to show that there are no leaks, and often running into the problem of complex enforcement mechanisms and false alarms), or not track them (which reduces false alarms, but provides weak or no security guarantees).

This paper distinguishes between malicious and nonmalicious code. The attacker may exploit implicit flows with malicious code, and so they should be tracked. We show how this can be done by a security type system and by a monitor. For nonmalicious code, we explore a middle ground between the two extremes. We observe that implicit flows are often harmless in nonmalicious code: they cannot be exploited to efficiently leak secrets. To this end, we are able to guarantee strong information-flow properties with a combination of an explicit-flow and a graph-pattern analyses. Initial studies of industrial code (secure logging and data sanitization) suggest that our approach has potential of offering a desired combination of a lightweight analysis, strong security guarantees, and no excessive false alarms.

Keywords. Security, noninterference, implicit flows, static analysis, monitoring.

1. Introduction

Information-flow technology is a promising approach for ensuring security by design and construction [27]. This technology helps controlling the flow of information between data of different levels of sensitivity. It is useful in the context of *confidentiality*, when secret inputs should not be leaked to public outputs, and *integrity*, when untrusted inputs should not affect trusted outputs.

Malicious vs. nonmalicious code Information-flow techniques are useful for both finding security problems in potentially malicious code, such as JavaScript code on untrusted (or attacked) web pages (e.g., [31]), or untrusted Java applets (e.g., [12]) and in nonmalicious code, such as server-side code (e.g., [8]). In the former scenario, the attacker has full control over the code. The attacker's goal is to craft the code and input data in order to circumvent the security mechanisms and leak and/or corrupt sensitive information. In the latter scenario, the code is typically written without malice. However, the code might

contain a vulnerability (such as a forgotten user input validation). The attacker's goal is to craft input data in order to exploit the vulnerability and leak and/or corrupt sensitive information. We assume attackers that can observe public outputs and provide corrupt data. For the purpose of this work, we ignore more powerful attackers capable to measure time, power consumption, or others covert channels [20].

Similar techniques have been used in the literature for both scenarios. In this paper, we observe that some code patterns are uncommon in the latter scenario, which opens possibilities for lightweight and effective analyses.

Explicit vs. implicit flows There has been much recent progress on understanding information flow in languages of increasing complexity [27,24,30,26]. When tracking information flow, of particular concern are *implicit* flows, i.e., flows through control flow when computation branches on secret data and performs publicly observed side effects depending on which branch is taken. This is as opposed to *explicit* flows, where secret data is directly passed to public containers.

One example of an implicit flow can be produced due to an unhandled exception. If computation involving sensitive data may raise an exception, this might result in an insecure information flow. However, experiments by King et al. [19] suggest that exception-related annotations are not always critical for security. For example, they find that 706 out of 757 unhandled exception warnings of JLift [18], an interprocedural extension of Jif [24] (information-flow analyzer for Java), are in fact false alarms (around 93%!).

Two extremes The large body of literature exercises two extreme views on implicit flows: either track or ignore them. The former view originates from work of Denning and Denning [13], which has been revived in the context of security for mobile code [33] (see an overview [27] for this line of work). As of now, the state-of-the-art information-flow tools such as FlowCaml [25,30], the SPARK Examiner [5,7,26], and Jif [23,24] (along with its extensions Sif [9], SWIFT [8], JLift [18]) track implicit flows.

On the other hand, several programming languages, such as Perl, PHP, and Ruby, support a *taint* mode, which is an information-flow tracking mechanism for integrity. The taint mode treats input data as untrusted and propagates the taint labels along the computation so that tainted data cannot directly affect sensitive operations. This mechanism tracks explicit but not implicit flows. Similarly, static approaches (e.g., [29,14]) only track explicit flows in the context of input validation.

Why would one want to track implicit flows in the light of the above experiments suggesting that most of them are false alarms? The reason is that implicit flows may result in leaking secrets and compromising sensitive data in an efficient way (discussed below). The price is, however, rather high in terms of usability: tight control on side effects (including those related to exceptions) needs to be enforced.

The other extreme (ignoring implicit flows) is not always settling either. Giving up security for usability may be often inappropriate.

Middle ground We argue that the attacker may exploit implicit flows with malicious code, and so they should be tracked. We show how this can be done by a security type system [33] and a monitor [28] without excessive false alarms. For nonmalicious code, we explore a middle ground between the two extremes. We observe that for nonmalicious code, implicit flows are often harmless: they cannot be exploited to efficiently leak secrets. How do we draw a formal line between harmless and dangerous implicit flows? Consider a typical example of an implicit flow:

$$\texttt{if } \mathit{secret} \texttt{ then } \mathit{public} := 1 \texttt{ else } \mathit{public} := 0 \qquad \text{(Impl)}$$

Depending on the secret value of variable secret, the public value of variable public is assigned 1 or 0. Is the flow in this example dangerous? The key is the *context*, where this code fragment appears. If the context is empty, i.e., the above fragment is the program, then a run of the program leaks at most one bit of information about the secret. This can be an acceptable leak. On the other hand, if the above fragment happens to be in the following context:

$$\begin{array}{l} l := 0; \\ \texttt{while } (n \geq 0) \texttt{ do} \\ \quad k := 2^{n-1}; \\ \quad \mathit{secret} := h \geq k; \\ \quad \texttt{if } \mathit{secret} \texttt{ then } \mathit{public} := \mathit{1} \texttt{ else } \mathit{public} := 0; \\ \quad \texttt{if } \mathit{public} \texttt{ then } h := h - k; l := l + k \texttt{ else skip}; \\ \quad n := n - 1 \end{array} \qquad \text{(Mag)}$$

then the program leaks bit-by-bit the secret variable h to public variable l (assuming h is an n-bit nonnegative integer and all variables except for secret and h are public). This example shows that implicit flows can be *magnified* by loops, where a one-bit leak is turned into an n-bit leak in linear time in the size of the secret (n).

We observe that magnified leaks are rather unusual in nonmalicious code. This enables us to guarantee strong information-flow properties with a simple analysis technique. This technique involves explicit-flow analysis to prevent explicit flows, as in $\mathit{public} := \mathit{secret}$. As mentioned above, such techniques have been successfully developed for such languages as Perl, PHP, and Ruby. In addition, we have lightweight implicit-flow analysis: (i) we make sure there are no cyclic patterns in the control-flow graph (CFG) of the program that involve branching on secrets (or that there are such patterns but they can be unfolded into conditionals), and (ii) we make a count on how many conditionals there are that branch on secrets. The number is a bound on how many bits are leaked by the program.

We report results from initial studies of industrial code (secure logging and data sanitization in the context of cross-site scripting vulnerability prevention). Our findings suggest that our approach has potential of offering a desired combination of a lightweight analysis, strong security guarantees, and no excessive false alarms.

Language independence The separation of the security analysis into code-based explicit-flow analysis and graph-based implicit-flow analysis provides the benefit of language-dependence for the latter. Indeed, to perform implicit-flow analysis, the main bulk of work is to generate the control-flow graph. This can be done by standard tools (we demonstrate this in Section 7 when we deal with Java programs). The pattern analysis of the graph is language-independent. A single implementation can be reused for all languages.

Contributions In brief, the paper offers the following contributions:

- Insights on distinguishing malicious and nonmalicious code opening up opportunities for lightweight and effective security analysis of the latter.

- Insights on explicit and implicit flows opening up for specialized analysis of the latter featuring soundness, permissiveness, and low rate of false alarms.
- An exploration of a middle ground between the two extreme views on policies: no information flow and only preventing explicit information flow.
- Formalization of the insights above to show how to guarantee strong information-flow properties with a combination of an explicit-flow and a language-independent graph-pattern analyses.
- Initial experiments with industrial code suggesting we have a desired combination of a lightweight analysis, strong security guarantees, and no excessive false alarms.

Organization The paper is organized as follows. Section 2 presents semantics of a simple imperative language. Sections 3–5 focus on the malicious-code scenario, following our earlier work [28] and borrowing some of its exposition. Sections 3 and 4 present a static and dynamic enforcement mechanisms for tracking explicit and implicit flows. These mechanisms correspond to a type system by Volpano et al. [33] and a monitor by Sabelfeld and Russo [28], respectively. Section 5 recapitulates results [33,28] on the soundness of the type system and monitor, as well as on the relative permissiveness. Section 6 focuses on the nonmalicious-code scenario. It describes a combination of an explicit-flow analysis and graph-pattern analysis that offers information-flow guarantees. Section 7 presents initial studies of industrial code: secure logging and data sanitization. Section 8 discusses related work. Section 9 offers some concluding remarks.

2. Semantics

For the purpose of demonstrating the ideas, we fix a simple imperative language. Figure 1 presents the semantics for the language. Configurations have the form $\langle c, m\rangle$, where c is a *command* and m is a *memory* mapping variables to values. Semantic rules have the form $\langle c, m\rangle \xrightarrow{\alpha} \langle c', m'\rangle$, which corresponds to a small step between configurations. If a transition leads to a configuration with the special command $stop$ and some memory m, then we say the execution *terminates* in m. Observe that there are no transitions triggered by $stop$. The special command end signifies exiting the scope of an `if` or a `while`. Observe that end is executed after the branches of those commands. Commands $stop$ and end can be generated during execution of programs but they are not used in initial configurations, i.e., they are not accessible to programmers. For simplicity, we consider simple integer expressions in our language (i.e., constants, binary operations, and variables). The semantics for expressions is then standard and thus we omit it here. We denote the result of evaluating expression e under memory m as $m(e)$. The semantics are decorated with *events* α for communicating program events to an execution monitor.

Event nop signals that the program performs a `skip`. Event $a(x, e)$ records that the program assigns the value of e in the current memory to variable x. Event $b(e)$ indicates that the program branches on expression e. Finally, event f is generated when the structure block of a conditional or loop has finished evaluation.

Assume $cfg, cfg', \ldots$ range over command configurations and $cfgm, cfgm', \ldots$ range over monitor configurations. For this work, it is enough to think of monitor configurations as simple stacks of security levels (see below). The semantics are paramet-

$$\langle \texttt{skip}, m\rangle \xrightarrow{nop} \langle stop, m\rangle \qquad \frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{a(x,e)} \langle stop, m[x \mapsto v]\rangle}$$

$$\frac{\langle c_1, m\rangle \xrightarrow{\alpha} \langle stop, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\alpha} \langle c_2, m'\rangle} \qquad \frac{\langle c_1, m\rangle \xrightarrow{\alpha} \langle c_1', m'\rangle \qquad c_1' \neq stop}{\langle c_1; c_2, m\rangle \xrightarrow{\alpha} \langle c_1'; c_2, m'\rangle}$$

$$\frac{m(e) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{b(e)} \langle c_1; end, m\rangle}$$

$$\frac{m(e) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{b(e)} \langle c_2; end, m\rangle}$$

$$\frac{m(e) \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \xrightarrow{b(e)} \langle c; end; \texttt{while}\ e\ \texttt{do}\ c, m\rangle} \qquad \frac{m(e) = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \xrightarrow{b(e)} \langle end, m\rangle}$$

$$\langle end, m\rangle \xrightarrow{f} \langle stop, m\rangle$$

Figure 1. Command semantics

$$pc \vdash \texttt{skip} \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq \Gamma(x)}{pc \vdash x := e} \qquad \frac{pc \vdash c_1 \qquad pc \vdash c_2}{pc \vdash c_1; c_2}$$

$$\frac{lev(e) \sqcup pc \vdash c_1 \qquad lev(e) \sqcup pc \vdash c_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \frac{lev(e) \sqcup pc \vdash c}{pc \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

Figure 2. Typing rules

ric in the monitor μ, which is assumed to be described by transitions between monitor configurations in the form $cfgm \xrightarrow{\alpha}_{\mu} cfgm'$. The rule for monitored execution is:

$$\frac{cfg \xrightarrow{\alpha} cfg' \qquad cfgm \xrightarrow{\alpha}_{\mu} cfgm'}{\langle cfg \mid_{\mu} cfgm\rangle \longrightarrow \langle cfg' \mid_{\mu} cfgm'\rangle}$$

The simplest example of a monitor is an all-accepting monitor μ_0, which is defined by $\epsilon \xrightarrow{\alpha}_{\mu_0} \epsilon$, where ϵ is its only state (the empty stack). This monitor indeed accepts all events α in the underlying program.

$$st \xrightarrow{nop} st \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{st \xrightarrow{a(x,e)} st} \qquad st \xrightarrow{b(e)} lev(e) : st$$

$$hd : st \xrightarrow{f} st \quad st \xrightarrow{b(e)} lev(e) : st$$

Figure 3. Monitoring rules

3. Type System

Figure 2 displays a Denning-style static analysis in the form of a security type system by Volpano et al. [33].

Typing environment Γ maps variables to security levels in a security lattice. For simplicity, we assume a security lattice with two levels L and H for low (public) and high (secret) security, where $L \sqsubset H$. Function $lev(e)$ returns H if there is a high variable in e and otherwise returns L.

Typing judgment for commands has the form $pc \vdash c$, where pc is a security level known as the *program counter* that keeps track of the context.

Explicit flows (as in $l := h$) are prevented by the typing rule for assignment that disallows assignments of high expressions to low variables.

Implicit flows (as in `if` h `then` $l := 1$ `else` $l := 0$) are prevented by the pc mechanism. It demands that when branching on a high expression, the branches must be typed under high pc, which prevents assignments to low variables in the branches.

4. Monitor

Figure 3 presents monitor μ_1 (we omit the subscript μ_1 in the transition rules for clarity). The monitor either accepts an event generated by the program or blocks it by getting stuck. The monitor configuration st is a stack of security levels, intended to keep track of the current security context: the security levels of the guards of conditionals and loops whose body the computation currently visits. This is a dynamic version of the pc from the previous section. Event nop (that originates from a `skip`) is always accepted without changes in the monitor state. Event $a(x, e)$ (that originates from an assignment) is accepted without changes in the monitor state but with two conditions: (i) that the security level of expression e is no greater than the security level of variable x and (ii) that the highest security level in the context stack (denoted $lev(st)$ for a stack st) is no greater than the security level of variable x. The former prevents *explicit* flows of the form $l := h$, whereas the latter prevents implicit flows of the form `if` h `then` $l := 1$ `else` $l := 0$, where depending on the high guard, the execution of the program leads to different low events.

Events $b(e)$ result in pushing the security level of e onto the stack of the monitor. This is a part of implicit-flow prevention: runs of program `if` h `then` $l := 1$ `else` $l := 0$ are blocked before performing an assignment l because the level of the stack is high when reaching the execution of the assignment. The stack structure avoids overrestrictive enforcement. For example, runs of program (`if` h `then` $h := 1$ `else` $h := 0$); $l := 1$ are

allowed. This is because by the time the assignment to l is reached, the execution has left the high context: the high security level has been popped from the stack in response to event f, which the program generates on exiting the `if`.

We have seen that runs of programs like `if` h `then` $l := 1$ `else` $l := 0$ are rejected by the monitor. But what about a program like `if` h `then` $l := 1$ `else skip`, a common example for illustrating that dynamic information-flow enforcement is delicate? If h is nonzero, the monitor blocks the execution. However, if h is 0, the program proceeds normally. Are we accepting an insecure program? It turns out that the slight difference between unmonitored and monitored runs (blocking in case h is nonzero) is sufficient for termination-insensitive security. In effect, the monitor prevents implicit flows by *collapsing the implicit-flow channel into the termination channel*; it does not introduce any more bandwidth than what the termination channel already permits. Indeed, implicit flows in unmonitored runs can be magnified by a loop so that secrets can be leaked bit-by-bit in linear time in the size of the secret. On the other hand, implicit flows in monitored runs cannot be magnified because execution is blocked whenever it attempts entering a branch with a public side effect. For example, one implication for uniformly-distributed secrets is that they cannot be leaked on the termination channel in polynomial time [3].

5. Security and permissiveness of implicit flow tracking

This section presents the formal results on tracking implicit flows by the type system and monitor. We assume μ_0 is the monitor that accepts all program events, and μ_1 is the monitor from Section 4. First, we show that the monitor μ_1 is strictly more permissive than the type system. If a program is typable, then all of its runs are not modified by the monitor. The appendix contains the details of all proofs.

Theorem 1 *If* $pc \vdash c$ *and* $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle$*, then* $\langle\langle c, m\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_1} st'\rangle$.

Proof. We prove a generalization of the theorem. Intuitively, the theorem holds because (i) the requirements for assignments in the type system and the monitor μ_1 are essentially the same; and (ii) there is a tight relation between the join operations for pc and pushing security levels on the stack st. □

Further, there are programs (e.g., `if` $l > l$ `then` $l := h$ `else skip`) whose runs are always accepted by the monitor, but which are rejected by the type system. Hence, the monitor is strictly more permissive than the type system.

We now show that both the type system and monitor enforce the same security condition: termination-insensitive noninterference [33]. Two memories m_1 and m_2 are *low-equal* (written $m_1 =_L m_2$) if they agree on the low variables. Termination-insensitive noninterference demands that starting with two low-equal initial memories, two terminating runs of a typable program result in low-equal final memories.

Theorem 2 *If* $pc \vdash c$*, then for all* m_1 *and* m_2*, where* $m_1 =_L m_2$*, whenever we have* $\langle\langle c, m_1\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_0} \epsilon\rangle$ *and* $\langle\langle c, m_2\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_0} \epsilon\rangle$*, then* $m_1' =_L m_2'$.

Proof. By adjusting the soundness proof by Volpano et al. [33]. □

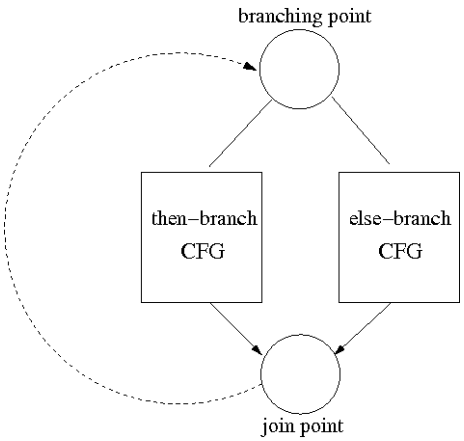

Figure 4. Control-flow graphs that presents the magnification pattern

Termination-insensitive noninterference also holds for the runs monitored by the monitor from Section 4:

Theorem 3 *For all* m_1 *and* m_2*, where* $m_1 =_L m_2$*, whenever* c *contains no* end *commands and* $\langle\langle c, m_1\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ *and* $\langle\langle c, m_2\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle$*, then* $m_1' =_L m_2'$*.*

Proof. By induction on $\longrightarrow^*$. □

6. Middle ground

In this section we explore the middle ground for implicit-flow analysis. We show how to achieve information-flow properties with a combination of an explicit-flow and a graph-pattern analyses.

We call *high branching* (*low branching*) branching instructions whose guards contain secret information (only public information). For example, high/low branching takes places in *high/low conditionals and loops*, respectively. In general, branching instructions may also come from exceptions and other constructs that introduce alternatives in programs' control-flow path. We start by defining a control-flow graph pattern that captures implicit flows that may be magnified.

Definition 1 (Magnification pattern) *Given a control-flow graph, a* magnification pattern *consists of high branching contained inside of a loop.*

Figure 4 shows a visual representation of this definition. The figure contains a branching instruction with two alternatives, the then and else branches. After one of the branches is executed, the control flow returns to the join point of the branch. We only consider languages that are well-structured, i.e., languages with no arbitrary jumps. After reaching the join point, some other instructions might be executed. What is important is that the program presents a loop that returns to the branching point. The following example illustrates what kind of programs present this pattern for a simple while language.

Example 1 *The following programs present the magnification pattern described in Definition 4.*

$$c_1 : \texttt{while}\ l \geq 0\ \texttt{do}\ (\ldots; \texttt{if}\ h\ \texttt{then}\ c_t\ \texttt{else}\ c_f; \ldots)$$

$$c_2 : \texttt{while}\ h \geq 0\ \texttt{do}\ \ldots$$

The dots correspond to the instructions that are not relevant for the magnification pattern. Due to the structure of c_1 and c_2, these programs might magnify implicit flows, i.e., implicit flows might not be harmless. Observe that program Mag, given in Section 1, presents a similar structure as c_1.

If a program does not contain magnification patterns, it does not mean that the implicit flows found there do not leak information. In fact, it might be possible that programs include as many implicit flows as the bit size of the secret. In this case, the secret might be leaked in linear time in the size of the secret.

Our main result in this section establishes that if a program neither contains explicit flows nor the magnification pattern, then it needs to branch on secrets at least k times in order to leak k bits of a secret. In this manner, we bound the number of leaked bits.

We now define the notion of *execution trees* for programs. These trees represent possible executions of programs when the low part of the initial memory is fixed. Intuitively, these trees describe how assignments can be performed by programs before branching on secrets.

Definition 2 (Execution trees) *Given a program c that does not present the magnification pattern, we define an* execution tree *for c as a binary tree where each node represents the sequence of assignments executed before branching on secrets.*

Intuitively, the internal nodes of an execution tree represent the assignments gathered by the program before branching on secrets. The leaves, on the other hand, represent all the assignments performed by the program when following a particular control-flow path. This definition resembles the notion of *decision trees* in the area of algorithms [4].

The following lemma establishes a relation between the number of leaked bits and the number of leaves in an execution tree.

Proposition 1 *Given a program c that contains no explicit flows or the magnification pattern, if c leaks at least k bits about the initial values of high variables once the public part of the initial memory is fixed, then the execution tree for c contains at least 2^k leaves.*

This leads us to the following theorem.

Theorem 4 *If a program c contains no explicit flows or the magnification pattern and leaks at least k bits about the initial values of high variables once the public part of the initial memory is fixed, then c contains at least k high branching instructions.*

Corollary 1 *If a program c contains k high branching instructions and contains no explicit flows or magnification patterns, then the program leaks at most k bits about the initial values of high variables once fix the public inputs.*

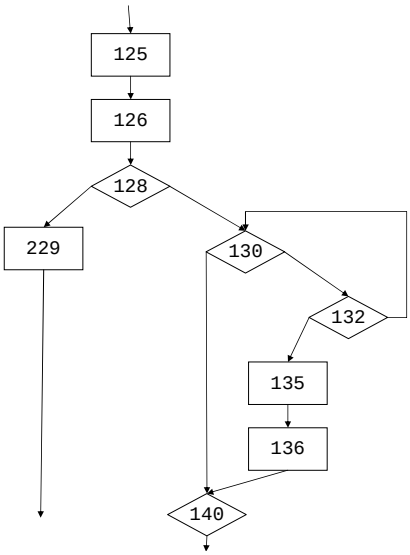

Figure 5. An example of a control-flow graph

Proof. By contradiction. We assume that the program leaks $k+1$ bits. Then, by Theorem 4, c contains at least $k+1$ high branching instruction, which gives us a contradiction. □

Declassification and endorsement We do not consider declassification (endorsement) policies for releasing (boosting the trust of) information. The security guarantees described above are valid for programs without such policies. Nevertheless, one possible direction to combine our results with declassification policies is in the line of work done by Hicks et al. [17], who consider that a piece of code between declassification points must satisfy noninterference. A scenario where our results can be applied consists of programs that always perform declassifications as the final step of their execution. We assume that the last line in the code is the only one including declassification. In this manner, our guarantees hold for the whole program but the last line. Similarly, our results can also be directly applied to scenarios where endorsement is performed as the first or last step of the computation.

7. Initial studies of industrial code

For an initial indication of the effectiveness of our approach, we have studied the code of industrial applications at SAP written in Java: secure logging and data sanitization in the context of cross-site scripting vulnerability prevention.

7.1. Magnification patterns in nonmalicious code

The purpose of the studies is to reveal whether magnification patterns are frequently present in nonmalicious code.

We do this examination in the following steps:

- Security levels are assigned to variables.

- Control-flow graphs are generated for methods using Control Flow Graph Factory plug-in from Dr. Garbage [1].
- We check every loop in the control-flow graph. If the loop guard contains secrets, then a magnification pattern is found. If the guard is low, we check if there exists branching on secrets in the loop body. If that is the case, a magnification pattern is found. Otherwise, the loop is not problematic.

As an example of the analyzed code, we present the following piece of anonymized source code.

```
...
IRL[] ls = getLs();
IRL targetRL = null;

if (ls != null)
{
  for (IRL rl : ls)
  {
    if (rl.implementAction
           (requestParameters))
    {
      // ...
      targetRL = rl;
      break;
    }
  }
}
...
```

The corresponding control-flow graph is depicted in Figure 5. In this example, there is a loop containing a branching instruction. Both the loop and the branching are not on variables with high security level. Therefore, they do not constitute a magnification pattern.

As an initial case study, we check a version of Java source code of a module in SAP's system. There are more than 30 classes analyzed. In total, there are more than 6000 lines of source code and more than 150 methods. With respect to branching instructions, there are around 420 low and around 60 high branching instructions. There is only one magnification pattern in this case study. The pattern arises in the piece of code when the user tries to log in into the system. The system only allows the user to try for at most three times. If the user is not able to provide valid credentials during the three times, his or her account will be locked. At first sight, this appears as a loop that involves a branch on secrets. However, since the maximum number of possible tries is bounded, this loop can be unfolded into a sequence of at most three conditionals. Therefore, this pattern is not dangerous (and the information leak is bounded by the number of conditionals in the unfolding: at most three bits).

7.2. Logging API and encryption API

Logging and tracing are important elements for securing application server systems. Logs are important for monitoring applications and to track events if problems occur, as well as for auditing the correct usage of the system.

The SAP Logging API is provided with all functionality for both tracing and event logging. The following methods are provided to write messages with different severity. They have intuitive names that indicate the severity levels such as `FATAL`, `ERROR`, `WARNING`, `INFO`, `PATH`, and `DEBUG`.

```
fatalT(string the_message) ;
errorT(string the_message) ;
warningT(string the_message) ;
infoT(string the_message) ;
pathT(string the_message) ;
debugT(string the_message) ;
```

For more information about SAP Logging API, please refer to [2].

One security policy is that before sensitive information is logged, it must be encrypted in order to prevent information leakage.

In SAP NetWeaver Platform, there are interfaces and classes derived from them available for implementing digital signatures and encryption in the applications. We now proceed to describe them.

The interface `ISsfData` is the central interface used for the cryptographic functions. Its underlying classes specify the data format used, for example, `SsfDataPKCS7`, `SsfDataSMIME` and `SsfDataXML`. The available methods are `sign`, `verify`, `encrypt`, `decrypt`, and `writeTo`.

The interface `ISsfProfile` provides access to the user's or server's profile, where the private key and corresponding public-key certificate are stored. If the public-key certificate has been signed by a *certification authority* (CA), then the interface also provides access to the CA chain associated with the certificate.

The interface `ISsfPab` contains a list of public-key certificates belonging to others. This public-key certificates contained in this list are used to verify their owners' digital signatures or to encrypt documents.

For more information about SAP interfaces and classes for using digital signatures and encryption, please refer to [2].

7.3. Security analysis for logging

As said before, one security consideration in SAP's system is that before sensitive information is logged, it must be encrypted in order to prevent information leakage. From the point of view of information-flow analysis, the variables containing sensitive information are assigned the high security level, while the log file is assigned the low security level. The only manner for high security level information to flow into the log file is through declassification.

To check the security of our logging system, we firstly assign security levels, i.e., $\Gamma(v)$, to each relevant variable v. This needs domain knowledge of the developer, and is done manually. Variables containing sensitive information, e.g., password and salary, are

assigned high security level. The variable associated with the log file is assigned a low security level. In the case studies, writing to the log file is implemented as follows.

```
Location myLoc = Location.
  getLocation("com.sap.fooPackage.FooClass");
myLoc.addLog(new ConsoleLog());
myLoc.warningT("Sample message" + password);
```

The last command is where sensitive information is written to the log file. This statement is a function call whose effect (in terms of explicit flows) is similar to the one of an assignment.

Encryption is implemented in the following way.

```
ISsfData data;
profile = new \\
  SsfProfileKeyStore(keyStore, alias, null);
result = data.encrypt(profile);
```

Observe that the last statement is where the declassification occurs. Variables `data` and `result` are assigned to the high and low security levels, respectively. This information flow is permitted.

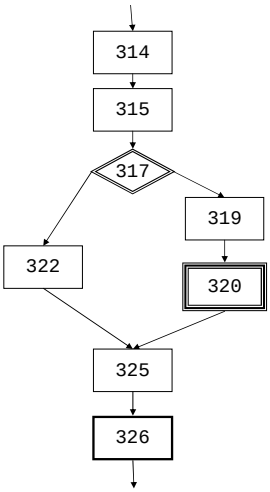

Figure 6. A control-flow graph of secure logging

The first step to analyze applications that use the logging API is to use the explicit flow analysis to check whether any secret data is directly placed into the log file as a plain text. We performed the analysis on a version of Java source code of the module in SAP's system. We found some instances, where sensitive information is logged before being encrypted. However, these flows were fixed in later versions of the code.

Having done that, we generate the corresponding control-flow graphs. In one source file of our case study, we found the piece of control-flow graph depicted in Figure 6. In the figure, line 320 is the declassification statement, and line 326 is where the log file is written. In addition to that, the branch, i.e., line 317, is on a variable with high security

level. According to the previous discussion, at most one bit of the sensitive information may be unintentionally leaked by the implicit flow.

7.4. SAP output encoding framework

Cross-site scripting (XSS) attacks may occur when a web application accepts data originating from a user and sends it to another user's browser without first validating or encoding it. For example, suppose an attacker embeds malicious JavaScript code into his or her profile on a social web site. If the site fails to validate such input, that code may execute malicious code in the browser of any other user who visits that profile.

In SAP NetWeaver Platform, the SAP Output Encoding Framework could be used to prevent XSS attacks. This applies when application developers generate HTML codes. By encoding user supplied input before rendering it, any inserted scripts are prevented from being transmitted to users in executable form. The encoding functions implement the corresponding sanitization routines.

In order to use SAP Output Encoding Framework to prevent XSS attacks, the following four different cases need to be distinguished.

Case 1: XSS attacks can occur as output between tags. In this case, the following functions should be applied for output encoding:

```
static String escapeToHTML(String input);
static String escapeToHTML
  (StringBuffer sb,
   String input, int maxLength);
static String escapeToHTML
  (String input, int maxLength);
```

Case 2: XSS attacks can occur as output inside tags, but output is not a URL or style. In this case, the functions named `escapeToAttributeValue` should be applied for output encoding.

Case 3: XSS attacks can occur as output which is a URL or style. In this case, the functions named `escapeToURL` should be applied for output encoding.

Case 4: XSS attacks can occur as output inside a SCRIPT context. In this case, the functions named `escapeToJS` should be applied for output encoding.

For more detailed information about the usage of SAP Output Encoding Framework, please refer to [2].

7.5. Security analysis for XSS prevention

Note that preventing XSS attacks is not about confidentiality but about integrity. The treatment of integrity is dual to confidentiality, and so our technique can also applies to this case.

In this case study, the user input is assigned a low-integrity level, and the output is assigned a high-integrity security level. The output encoding functions are considered as *endorsement*, a dual of declassification. The statement writing into output is considered as an assignment. By doing the adaptation, we can analyze the program in the same way as we did for the previous case study. We did not find any vulnerability with respect to XSS in this case study since data sanitization, and thus endorsement, is performed early in the code.

8. Related work

As mentioned previously, the large body of literature exercises two extreme views on implicit flows: either track or ignore them. The former view originates from work of Denning and Denning [13], which has been revived in the context of security for mobile code [33] (see an overview [27] for this line of work). As of now, the state-of-the-art information-flow tools such as FlowCaml [25,30], the SPARK Examiner [5,7,26], and Jif [23,24] (along with its extensions Sif [9], SWIFT [8], JLift [18]) track implicit flows.

On the other hand, several programming languages, such as Perl, PHP, and Ruby, support a *taint* mode, which is an information-flow tracking mechanism for integrity. The taint mode treats input data as untrusted and propagates the taint labels along the computation so that tainted data cannot directly affect sensitive operations. This mechanism (as well as others, designed for confidentiality [32]) tracks explicit but not implicit flows. Similarly, static approaches (e.g., [29,14]) only track explicit flows in the context of input validation.

In terms of policies, the former view corresponds to *noninterference* [15] that postulates that secret inputs may not affect public outputs of a program. The latter corresponds to *weak secrecy* [32] that postulates that no sequence of assignment commands that a given run executes leaks information. This condition ignores flows due to control flow. For example, the simple implicit flow program Impl from Section 1 is accepted by this condition, and so is the magnified attack Mag that leaks the secret in linear time.

As mentioned in Section 1, King et al. [19] suggest that exception-related annotations are not always critical for security. For example, they find that 706 out of 757 unhandled exception warnings of JLift [18], an interprocedural extension of Jif [24] (information-flow analyzer for Java), are in fact false alarms (around 93%!).

Chang et al. [6] investigate denial-of-service vulnerabilities. Interestingly, they have findings that are related to ours. They observe that a typical pattern of such a vulnerability is a loop, where the attacker may affect the guard. They present a static combination of control-flow and data dependency analyses for C programs.

Haack et al. [16] describe an approach to tracking explicit information flows in JML. The paper contains a general discussion on explicit vs. implicit flows and confidentiality vs. integrity. However, their primary focus is safety properties.

The line of work on giving quantitative bounds on how much information can be leaked by programs is close in spirit to ours. Clark et al [10] bound leakage in terms of an upper bound on the number of information-theoretic bits. Lowe [21] considers counting a number of equivalence classes in an equivalence-relation model, which can be used for an information-theoretic bound. Clarkson et al. [11] include belief into the analysis of quantitative information flow in a language-based setting. McCamant and Ernst [22] suggest a dynamic approach to giving a quantitative bound on the amount of information a program leaks during a run. Due to the dynamic nature of the latter analysis, it is more permissive than the static ones mentioned before. Hence, the rate of false alarms is more acceptable. However, a sacrifice is the runtime overhead and late discovery of insecurities.

9. Conclusion

We have reported insights on implicit flows in malicious vs. nonmalicious code. For nonmalicious code, it turns out that rather than performing fully-fledged information-flow analysis, we can give strong information-flow properties guarantees with a combination of an explicit-flow and a graph-pattern analyses.

We have presented evidence that the middle ground that we explore between the two extreme views (noninterference and weak secrecy) is a meaningful one.

Our studies of industrial code are feasibility studies: they have been mostly performed by hand (with the exception of control-flow graph generation tools). Future work involves implementing the rest of the components for the analysis: the explicit flow checker and the pattern finder/counter. Furthermore, more extensive case studies will help determine how common secret branching is in routine server code. Another concern about our approach is related to scalability. In particular, in languages with recursion and method calls, it is necessarty to generate a full-program control-flow graph. Optimizations that mitigate the cost of such analysis are worth exploring.

Acknowledgments

This work was funded by the Information Society Technologies programme of the European Commission under the IST-2005-015905 MOBIUS project and by the Swedish research agencies SSF and VR.

References

[1] Dr. garbage. `http://www.drgarbage.com`.

[2] Sap netweaver 7.0 knowledge center. `http://help.sap.com/content/documentation/netweaver/docu_nw_70_design.htm`.

[3] A. Askarov and S. Hunt and A. Sabelfeld and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, Oct. 2008.

[4] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[5] J. Barnes and J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[6] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[7] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.

[8] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, Oct. 2007.

[9] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, Aug. 2007.

[10] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[11] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 31–45, June 2005.

[12] M. Dam and P. Giambiagi. Information flow control for cryptographic applets. Presentation at the Dagstuhl Seminar on Language-Based Security, Oct. 2003. www.dagstuhl.de/03411/Materials/.

[13] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[14] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42?–51, 2002.

[15] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[16] C. Haack, E. Poll, and A. Schubert. Explicit information flow properties in JML. In *Proc. WISSEC*, 2008.

[17] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 65–74, June 2006.

[18] D. King. JLift. Software release. Located at http://www.cse.psu.edu/~dhking/jlift, 2008.

[19] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proc. International Conference on Information Systems Security (ICISS)*, volume 5352 of *LNCS*, pages 56–70. Springer-Verlag, Dec. 2008.

[20] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, Oct. 1973.

[21] G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.

[22] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 193–205, 2008.

[23] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[24] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001–2008.

[25] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.

[26] Praxis High Integrity Systems. SPARKAda Examinar. Software release. http://www.praxis-his.com/sparkada/.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[28] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[29] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. USENIX Security Symposium*, pages 201?–220, 2001.

[30] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.

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

[32] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.

[33] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

A. Appendix

We introduce some auxiliary lemmas. We describe the most important ones here. We start by showing lemmas related to sequential composition of monitored executions.

Lemma 1 *If* $\langle\langle c, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_\mu st'\rangle$*, then* $st = st'$*, where* $\mu \in \{\mu_0, \mu_1\}$.

Lemma 2 *Given that* $stop; c'$ *denotes* c'*, if* $\langle\langle c_1, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_\mu st'\rangle$*, then* $\langle\langle c_1; c_2, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c'; c_2, m'\rangle \mid_\mu st'\rangle$*, where* $\mu \in \{\mu_0, \mu_1\}$.

Lemma 3 *If* $\langle\langle c_1; c_2, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_\mu st'\rangle$ *and* c_1 *contains no* end *instructions, then there exists* c^**,* m''*, and* st^* *such that* $c' = c^*; c_2$ *and* $\langle\langle c_1, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_\mu st^*\rangle$*; or* $\langle\langle c_1, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_\mu st\rangle$ *and* $\langle\langle c_2, m''\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_\mu st'\rangle$*, where* $\mu \in \{\mu_0, \mu_1\}$.

These lemmas can be proved by a simple induction on $\longrightarrow^*$. Before proving Theorem 1, we prove a generalization of it described in the following lemma.

Lemma 4 *If* $pc \vdash c$*,* $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle$*, then it holds* $\forall\ lev(st) \sqsubseteq pc \cdot \exists\ lev(st') \cdot \langle\langle c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_1} st'\rangle$.

Proof. By induction on $\longrightarrow^*$ and the number of sequential instructions in c. We only show the most interesting cases.

$x := e$**)** Given a st such that $lev(st) \sqsubseteq pc$, we need to prove that exists st' such that $lev(st')$ and $\langle\langle x := e, m\rangle \mid_{\mu_1} st\rangle \longrightarrow \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$. Let's take $st' = st$. Then, the transition under μ_1 is possible provided that $lev(e) \sqsubseteq \Gamma(x)$ and $lev(st) \sqsubseteq \Gamma(x)$. By the typing rules, it holds that $lev(e) \sqsubseteq \Gamma(x)$ and $pc \sqsubseteq \Gamma(x)$. By these two facts, and having that $lev(st) \sqsubseteq pc$, it holds that $lev(e) \sqsubseteq \Gamma(x)$ and $lev(st) \sqsubseteq \Gamma(x)$.

`if` e `then` c_1 `else` c_2**)** Let's assume that $m(e) \neq 0$ (the proof follows the same structure when $m(e) = 0$). We omit the proof when $\longrightarrow_0$ since it holds trivially. By semantics, we know that

$$\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow \langle\langle c_1; end, m\rangle \mid_{\mu_0} \epsilon\rangle \tag{1}$$

$$\langle\langle c_1; end, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle \tag{2}$$

By definition of the monitor, we know that

$$\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \mid_{\mu_1} st\rangle \longrightarrow \langle\langle c_1; end, m\rangle \mid_{\mu_1} lev(e) : st\rangle \tag{3}$$

If $\longrightarrow^*$ is $\longrightarrow_0$ in (2), the result follows from (3). Otherwise, by applying Lemma 3 on (2) and semantics, we have that there exists m'', c^*, and st^* such that

$c' = c^*; end$**)** In this case, we have that

$$\langle\langle c_1, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_{\mu_0} st^*\rangle \tag{4}$$

We know that $st^* = \epsilon$ from the definition of μ_0. We apply IH on $lev(e) \sqcup pc \vdash c_1$ (obtaining from the typing rules) and (4), then we obtain that $\forall\ lev(st_1) \sqsubseteq$

$lev(e) \sqcup pc \cdot \exists\, lev(st_1') \cdot \langle\langle c_1, m\rangle \mid_{\mu_1} st_1\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_{\mu_1} st_1'\rangle$. Let's instantiate this formula by taking $st_1 = lev(e) : st$. We then have that

$$\langle\langle c_1, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_{\mu_1} st_1'\rangle \tag{5}$$

By Lemma 2 applied to (5) and end, we obtain $\langle\langle c_1; end, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_1} st_1'\rangle$. The result follows from this transition and (3).

$c' \neq c^*; end$**)**

$$\langle\langle c_1, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_{\mu_0} \epsilon\rangle \tag{6}$$

$$\langle\langle end, m''\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle \tag{7}$$

By IH on $lev(e) \sqcup pc \vdash c_1$ (obtaining from the typing rules) and (6), we have that $\forall\, lev(st_1) \sqsubseteq lev(e) \sqcup pc \cdot \exists\, lev(st_1') \cdot \langle\langle c_1, m\rangle \mid_{\mu_1} st_1\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_{\mu_1} st_1'\rangle$. Let's instantiate this formula with $st_1 = lev(e) : st$. We then have that

$$\langle\langle c_1, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_{\mu_1} st_1'\rangle \tag{8}$$

At this point, we do not know the shape of st_1', but we can deduced it by applying the Lemma 1 to it: $st_1' = lev(e) : st$. Then, by Lemma 2 on (8) and semantics for end, we have that

$$\langle\langle c_1; end, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle end, m''\rangle \mid_{\mu_1} lev(e) : st\rangle \tag{9}$$

In the case that $\longrightarrow^*$ is $\longrightarrow_0$ in (7), the result holds from (3) and (9). Otherwise, from semantics rules in (7), we know that $c' = stop$ and $m' = m''$. By monitor semantics, we know that

$$\langle\langle end, m''\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow \langle\langle stop, m''\rangle \mid_{\mu_1} st\rangle \tag{10}$$

The result then follows from (3), (9), and (10).

`while` e `do` c**)** Similar to the previous case.

□

We can then prove the first theorem.

Theorem 1 *If* $pc \vdash c$ *and* $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_0} \epsilon\rangle$*, then* $\langle\langle c, m\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$.

Proof. By Lemma 4, we obtain that $\forall\, lev(st) \sqsubseteq pc \cdot \exists\, lev(st') \cdot \langle\langle c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$. The result follows by instantiating the formula with $st = \epsilon$ since $lev(\epsilon) = L$. □

To prove Theorem 2, we firstly prove that, for terminating programs, there is an isomorphism between the command semantics and executions under μ_0.

Lemma 5 *Given command* c *that contains no* end *instructions,* $\langle c, m\rangle \longrightarrow^* \langle stop, m'\rangle \Leftrightarrow \langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_0} \epsilon\rangle$.

Proof. Both directions of the implication are proved by a simple induction on $\longrightarrow^*$. □

Now, we are in conditions to prove the mentioned Theorem.

Theorem 2 *If $pc \vdash c$, then for all m_1 and m_2, where $m_1 =_L m_2$, whenever we have $\langle\langle c, m_1\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_0} \epsilon\rangle$ and $\langle\langle c, m_2\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_0} \epsilon\rangle$, then $m_1' =_L m_2'$.*

Proof. By Lemma 5, we have that $\langle c, m_1\rangle \longrightarrow^* \langle stop, m_1'\rangle$ and $\langle c, m_2\rangle \longrightarrow^* \langle stop, m_2'\rangle$. The result follows by applying the soundness theorem from [33] to $pc \vdash c$, $\langle c, m_1\rangle \longrightarrow^* \langle stop, m_1'\rangle$, and $\langle c, m_2\rangle \longrightarrow^* \langle stop, m_2'\rangle$. □

We need two auxiliary lemmas in order to prove Theorem 3. They express that public variables cannot be affected when the security level of the monitor's stack is H.

Lemma 6 *If c contains no end instructions, $lev(st) = H$, and $\langle\langle c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$, then $m =_L m'$.*

Proof. By induction on $\longrightarrow^*$. □

Lemma 7 *If c contains no end instructions, and $\langle\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$, then $m =_L m'$.*

Proof. By performing one small-step in the semantics and then applying Lemma 6. □

The next lemma is a generalization of Theorem 3.

Lemma 8 *For all m_1 and m_2, where $m_1 =_L m_2$, whenever c contains no end commands and $\langle\langle c, m_1\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ and $\langle\langle c, m_2\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle$, then $m_1' =_L m_2'$.*

Proof. By induction on $\longrightarrow^*$. We list the most interesting cases.

`if` e `then` c_1 `else` c_2) We consider the case when $lev(e) = H$ and that $m_1(e) \neq m_2(e)$. Otherwise, the proof follows by simply applying IH and Lemmas 2 and 3. We assume, without loosing generality, that $m_1(e) \neq 0$. Consequently, by semantics, we have that

$$\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m_1\rangle \mid_{\mu_1} st\rangle \longrightarrow$$

$$\langle\langle c_1; end, m_1\rangle \mid_{\mu_1} lev(e) : st\rangle \tag{11}$$

$$\langle\langle c_1; end, m_1\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle \tag{12}$$

$$\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m_2\rangle \mid_{\mu_1} st\rangle \longrightarrow$$

$$\langle\langle c_2; end, m_2\rangle \mid_{\mu_1} lev(e) : st\rangle \tag{13}$$

$$\langle\langle c_2; end, m_2\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle \tag{14}$$

By applying Lemma 3 on (12) and (14), we have that there exists m_1'' and m_2'' such that

$$\langle\langle c_1, m_1\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_1''\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (15)$$

$$\langle\langle end, m_1''\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle \quad (16)$$

$$\langle\langle c_2, m_2\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_2''\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (17)$$

$$\langle\langle end, m_2''\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle \quad (18)$$

By applying Lemma 6 on (15) and (17), we have that $m_1'' =_L m_1 =_L m_2 =_L m_2''$. By semantics, (16), and (18), we have that $m_1' = m_1''$ and $m_2' = m_2''$. Consequently, we have that $m_1' =_L m_2'$ as expected.

`while` e `do` c) The proof proceeds similarly as the previous case but also applying Lemma 7 when needed.

□

Theorem 3 *For all m_1 and m_2, where $m_1 =_L m_2$, whenever c contains no end commands and $\langle\langle c, m_1\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ and $\langle\langle c, m_2\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle$, then $m_1' =_L m_2'$.*

Proof. By applying Lemma 8 with $st = \epsilon$. □

Proposition 1 *Given a program c that contains no explicit flows or the magnification pattern, if c leaks at least k bits about the initial values of high variables once the public part of the initial memory is fixed, then the execution tree for c contains at least 2^k leaves.*

Proof. By contradiction. We assume that T has less than 2^k leaves. On one hand, we have 2^k possible initial memories that are different in the high part to account for the leaked secret. We denote the fixed low part of the memory, as described in the statement of the theorem, by m_L. Since the program c leaks k bits, there are at least 2^k final memories different in the low part. Observe that if an assignment $x := e$ is executed twice in a program, we consider them as different.

The execution tree T for c and m_L represents all the possible assignments that c can perform when running c with different secret values. Since we have 2^k possible final values for the leaked secret and T has less than 2^k leaves, the pigeonhole principle indicates that there exist two memories with initial high parts m_{H_1} and m_{H_2} such that m_{H_1} and m_{H_2} are different in the leaked secret and the execution of program c under these memories produces the assignments described by a given path in T. Let us sequentially compose all these assignments into the program c''. Observe that c'' might contain an infinite sequence of assignments due to the possible presence of infinite loops with low guards in c. Observe that c has no explicit flows by hypothesis and thus neither does c''. Denote (m_L, m_H) for the memory produced from the low and high projections m_L and m_H, respectively. Assuming that the sequence of assignments in c'' is finite, we have that $\langle c, (m_L, m_{H_1})\rangle \Downarrow m^*$ and $\langle c'', (m_L, m_{H_1})\rangle \Downarrow m^*$, while $\langle c, (m_L, m_{H_2})\rangle \Downarrow m^{**}$ and $\langle c'', (m_L, m_{H_2})\rangle \Downarrow m^{**}$, where $m_L^* \neq m_L^{**}$. However, $m_L^* = m_L^{**}$ because c'' contains no explicit flows. We then reach a contradiction.

In the case that c'' contains an infinite sequence of assignments, e.g., in the presence of nonterminating loops with low guards, we have that for any prefix of the infinite sequence of assignments in c'', written c_s'', it holds that c_s'' contains no explicit flows. As a consequence, we have that $\langle c_s'', (m_L, m_{H_1})\rangle \Downarrow m^*$ and $\langle c_s'', (m_L, m_{H_2})\rangle \Downarrow m^{**}$, where

$m_L^* = m_L^{**}$. This fact contradicts that c'' leaks k bits. Observe that every prefix of c'' leads to low-equal memories. □

Theorem 4 *If a program c contains no explicit flows or the magnification pattern and leaks at least k bits about the initial values of high variables once the public part of the initial memory is fixed, then c contains at least k high branching instructions.*

Proof. Take an initial memory m such that when running program c under m, it leaks at least k bits about the initial values of high variables . Let us take the low projection of memory m, written m_L, consisting of the values of the low variables. Then, we construct the execution tree T for c. Let us assume now that c has strictly less than k high branching instructions and aim at reaching a contradiction. By Proposition 1, T has at least 2^k leaves. The question is now what is the least possible number of executed branches in T. This is achieved when T is most balanced. Since a binary tree with 2^k leaves must be at least of height k, k is the least number of possible high branching instructions in T. Thus, c contains at least k high branching instructions. □