We are Family: Relating Information-Flow Trackers (Extended Version)

Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld

Chalmers University of Technology, Gothenburg, Sweden

Abstract. While information-flow security is a well-established area, there is an unsettling gap between heavyweight *information-flow control*, with formal guarantees yet limited practical impact, and lightweight *tainting* techniques, useful for bug finding yet lacking formal assurance. This paper proposes a framework for exploring the middle ground in the range of enforcement from tainting (tracking data flows only) to fully-fledged information-flow control (tracking both data and control flows). We formally illustrate the trade-offs between the soundness and permissiveness that the framework allows to achieve. The framework is deployed in a staged fashion, statically embedding a dynamic monitor, being parametric in security policies, as they do not need to be fixed until the final deployment. This flexibility facilitates a secure app store architecture, where the static stage of verification is performed by the app store and the dynamic stage is deployed on the client. To illustrate the practicality of the framework, we implement our approach for a core of Java and evaluate it on a use case with enforcing privacy policies in the Android setting. We also show how a state-of-the-art dynamic monitor for JavaScript can be easily adapted to implement our approach.

Keywords: Language-based security · Information-flow control · Taint Tracking

1 Introduction

Motivation The sheer bulk of sensitive information that software manipulates makes security a major concern. A recent report shows that several of the top 10 most popular flashlight apps on the Google Play store may send sensitive information such as pictures and video, users' location, and the list of contacts, to untrusted servers [51]. Unfortunately, trusted code also incurs serious security flaws, as proven by the Heartbleed bug [53] found in the OpenSSL library.

Information-flow control [44] offers an appealing approach to security assurance by design. It helps tracking the flow of information from confidential/untrusted sources to public/trusted sinks, ensuring, for *confidentiality*, that confidential inputs are not leaked to public outputs, and, for *integrity*, that untrusted inputs do not affect trusted outputs.

Background Applications can leak information through programming-language constructs, giving rise to two basic types of information flows: *explicit* and *implicit* flows [21]. Consider a setting with variables *secret* and *public* for storing confidential (or *high*) and public (or *low*) information, respectively. Explicit flows occur whenever sensitive information is passed explicitly by an assignment, e.g., as in $public := secret$. Implicit flows arise via control-flow structures of programs, e.g. conditionals and loops, as in **if** $secret$ **then** $public := 0$ **else** $public := 1$. The final value of *public* depends on the initial value of *secret* because of a *low assignment*, i.e., assignment to a low variable, made in a *high context*, i.e., branch of a conditional with a secret guard.

Information-flow control is typically categorized as static and dynamic: (1) *Static* techniques mainly impose Dennings' approach [21] by assigning security labels to input data, e.g. variables, APIs, and ensuring separation between secret and public computation, essentially by maintaining the invariant that no low assignment [58,44,32] occurs in a high context. Other static techniques include program logics [10,13], model checking [8,23], abstract interpretations [27] and theorem proving [20,40]. However, static techniques face *precision* (high false-positive rate) challenges, rejecting many secure programs. These challenges include *dynamic code evaluation* and *aliasing*, as illustrated by the snippet $x.f := 0$; $y.f := secret$; $\textbf{out}(\mathbf{L}, x.f)$. A non-trivial static analysis would have to approximate whether object references x and y are aliases. Moreover, the fact that security policies are to be known at verification time makes them less suitable in dynamic contexts. (2) *Dynamic* techniques use program runtime information to track information flows [26,5,43]. The execution of the analyzed program is monitored for security violations. Broadly, the monitor enforces the invariant that no assignment from high to low variables occurs either explicitly or implicitly. Dynamic techniques are particularly useful in highly dynamic contexts and policies, where the code is often unknown until runtime. However, since the underlying semantic condition, *noninterference* [28], is not a trace property [38], dynamic techniques face challenges with branches not taken by the current execution. Consider the secure program that manipulates *location* information: **if** $(MIN \leq loc)$ && $(loc \leq MAX)$ **then** $tmp := loc$ **else skip**. If the user's (secret) location *loc* is within an area bound by constants *MIN* and MAX, the program stores the exact location in a temporary variable *tmp*, without ever sending it to a public observer. A dynamic analysis, e.g. No-Sensitive Upgrade [60,5], incorrectly rejects the program (due to a security label upgrade in a high context), although neither *loc* nor *tmp* are ever sent to an attacker. Permissive Upgrade [6] increases precision, however, it will incorrectly rule out any secure program that subsequently branches on variable *tmp*.

Combining dynamic and static analysis, *hybrid* approaches have recently received increased attention [36,18,37,39,31]. While providing strong formal guarantees, to date the practical impact of all these approaches has been limited, largely due to low precision (or *permissiveness*). Moreover, static, dynamic, and hybrid information-flow analysis require knowledge of the control-flow graph to properly propagate the *program counter* security label that keeps track of the sensitivity of the context. This label is difficult to recover whenever code has undergone heavyweight optimization and obfuscation, e.g. to protect its intellectual property, or in presence of reflection.

In contrast, *taint tracking* is a practical success story in computer security, with many applications at all levels of hardware and software stack [49,47]. Taint tracking is a pure data dependency analysis that only tracks explicit flows. It is successful thanks to its lightweight nature, ignoring any control-flow dependencies that would be otherwise required for fully-fledged information-flow control. On the downside, taint tracking is mainly used as a bug finding technique, providing, with a few exceptions [59,47,48], no formal guarantees. Importantly, implicit flows may occur not only in malicious code [42,33], but also in trusted programs (written by a trusted programmer) [35,52,34,11].

These considerations point to an unsettling gap between heavyweight techniques for information-flow control, with formal guarantees yet limited practical impact, and lightweight tainting techniques that are useful for bug finding yet lacking formal assurance.

Approach By considering the trade-offs between *soundness* and *permissiveness*, this paper explores the middle ground, by a framework for a range of enforcement mechanisms from tainting to fully-fledged information-flow control. We address *trusted* and *malicious* code. However, we make a key distinction between two kinds of implicit flows: *observable* implicit flows and *hidden* implicit flows, borrowing the terminology of Staicu and Pradel [52]. Observable implicit flows arise whenever a variable is updated under a high security context and later output to an attacker. Not all implicit flows are, however, observable, since also the absence of a variable update can leak information (cf. Example 3); we call these hidden implicit flows. Tracking explicit flows and observable implicit flows raises the security bar for trusted code [52]. It allows for permissive, lightweight and purely dynamic enforcement in the spirit of taint tracking, yet providing higher security assurance. To evaluate soundness and permissiveness of the technique, we propose *observable secrecy*, a novel security condition that captures the essence of observable implicit flows. It helps us answer the question: "what is the security price we pay for having fewer false positives for useful programs"? We remark that the distinction between observable and hidden implicit flows is purely driven by ease of enforcement and permissiveness. Moreover, we leverage existing techniques and extend the framework to account for hidden implicit flows, thus addressing malicious code. We then present a family of flow-sensitive dynamic monitors that enforce a range of security policies by adapting a standard information-flow monitor from the literature [5,43].

The framework is deployed in a staged fashion. We statically embed dynamic monitors for (observable and/or hidden) implicit flows into the program code by lightweight program transformation, and leverage a dynamic taint tracker to enforce stronger policies. For malicious code, we use the *cross-copying* technique, originally proposed by Vachharajani et al. [55] for systems code, to transform hidden implicit flows into observable implicit flows. The transformations and soundness proofs for theorems can be found in the full version of the paper [14].

Secure App Store The flexibility of the approach on the policy and enforcement side facilitates a secure app store architecture, depicted in Figure 1. Developers deliver the code to the App Store, which computes sources and sinks, and leverages the control-flow graph to convert implicit flows into explicit flows. For trusted (non-malicious) apps, a lightweight transformation converting observable implicit flows into explicit may be sufficient, otherwise cross copying is needed. Subsequently, the App Store can perform code optimizations and obfuscations,

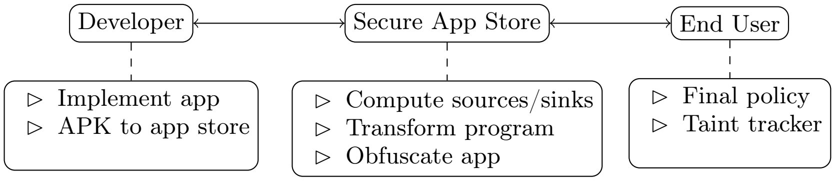

Fig. 1. Secure App Store Architecture

and publish the resulting APK file (together with sources and sinks) on behalf of the developer. Finally, end users can download the app, define their own security policies and run the app on a dynamic taint tracker, remarkably, with no need of the program's control-flow graph. Alternatively, end users can leverage static taint trackers [1,29] to verify their policies against the code.

We implement the transformations for a core of Java and evaluate them on the use case of a Pedometer app. We run the transformed app on TaintDroid [24] and check it against user-defined policies. We also show how JSFlow [30], a dynamic monitor for JavaScript, can provide higher precision by changing the security condition to observable secrecy.

Structure and Contributions In summary, the paper makes the following contributions: (*i*) observable secrecy, a security condition for validating soundness and precision wrt. observable implicit and explicit flows (Sect. 2); (*ii*) a framework that allows expressing a range of enforcement mechanisms from tainting to information-flow control (Sect. 3); (*iii*) lightweight transformations that leverage dynamic taint tracking for higher security assurance (Sect. 4); (*iv*) a flexible app store architecture and a prototype implementation for Android apps (Sect. 5).

2 Security Framework

We employ knowledge-based definitions [4,9,10] to introduce security conditions ranging from weak/explicit secrecy [59,47] to noninterference [28].

2.1 Language

Consider a simple imperative language with I/O primitives, SIMPL. The language expressions consist of variables $x \in Var$, built-in values $n \in Val$ such as integers and booleans, binary operators $\oplus$ and unary operators $\ominus$. We write **tt** for boolean value *true* and **ff** for boolean value *false*. The language constructs contain skip, assignment, conditional, loops, input and output. The full grammar of SIMPL can be found in Figure 2.

$$\begin{aligned} e &::= x \mid n \mid e \oplus e \mid \ominus e \\ P &::= \textbf{skip} \mid P; P \mid x := e \mid x \leftarrow \textbf{in}(\ell) \mid \textbf{out}(\ell, e) \\ &\quad\mid \textbf{if}\ e\ \textbf{then}\ P\ \textbf{else}\ P \mid \textbf{while}\ e\ \textbf{do}\ P \end{aligned}$$

Fig. 2. SIMPL Language Grammar

We use input and output channels to model communication of the program with the external world. We label input and output channels with *security levels* ℓ (defined below) that indicate the confidentiality level of the information transmitted on the corresponding channel. We denote the set of SIMPL programs by $\mathcal{P}$. We write $\bar{x}$ for a set of variables $\{x_1, \cdots, x_n\}$ such that for all $1 \le i \le n, x_i \in \mathit{Var}$, and $Vars(e)$ for the set of free variables of expression e.

We assume a bounded lattice of security levels $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$. A level $\ell \in \mathcal{L}$ represents the confidentiality of a piece of data present on a given channel or program variable. We assume that there is one channel for each security level $\ell \in \mathcal{L}$. As usual, $\sqsubseteq$ denotes the ordering relation between security levels and, $\sqcup$ and $\sqcap$ denote the *join* and *meet* lattice operators, respectively. We write $\top$ and $\bot$ to denote the top and the bottom element of the lattice. In the examples, we use a two-level security lattice $\mathcal{L} = \{\mathbf{L}, \mathbf{H}\}$ consisting of level $\mathbf{H}$ (high) for variables/channels containing confidential information and level $\mathbf{L}$ (low) for variables/channels containing public information, and $\mathbf{L} \sqsubseteq \mathbf{H}$. We focus on confidentiality, noting that integrity is similar through dualization [16].

We model input by environments $\mathcal{E} \in \mathit{Env}$ mapping channels to streams of input values. For simplicity, we consider one stream for each level $\ell \in \mathcal{L}$. An environment $\mathcal{E} : \mathcal{L} \to \mathbb{N} \to \mathit{Val}$ maps levels to infinite sequences of values. Two environments $\mathcal{E}_1$ and $\mathcal{E}_2$ are ℓ-equivalent, written $\mathcal{E}_1 \approx_\ell \mathcal{E}_2$, iff $\forall \ell'.\ \ell' \sqsubseteq \ell \Rightarrow \mathcal{E}_1(\ell') = \mathcal{E}_2(\ell')$. Another source of input are the initial values of program variables. We model memory as a mapping $m : \mathit{Var} \to \mathit{Val}$ from variables to values. We use $m, m_0, m_1, \ldots$ to range over memories. We write $m[x \mapsto n]$ to denote a memory m with variable x assigned the value n. We write $m(e)$ for the value of expression e in memory m. A *security environment* $\mathit{\Gamma} : \mathit{Var} \mapsto \mathcal{L}$ is a mapping from program variables to lattice elements. The security environment assigns security levels to the memory through program variables. We use the terms security *level* and security *label* as synonyms. Two memories m_1 and m_2 are ℓ-equivalent, written $m_1 \approx_\ell m_2$, iff $\forall x \in \mathit{Var}.\ \mathit{\Gamma}(x) \sqsubseteq \ell \Rightarrow m_1(x) = m_2(x)$.

An *observation* $\alpha \in \mathit{Obs}$ is a pair of a security level and a value, i.e. $\mathit{Obs} = \mathcal{L} \times \mathit{Val}$, or the empty observation ϵ. A *trace* τ is a finite sequence of observations. We write $\tau.\tau'$ for concatenation of traces τ and τ', and $|\tau|$ for the length of a trace τ. We denote by $\tau \upharpoonright_\ell$ the projection of trace τ at security level ℓ. Formally, we have $\epsilon \upharpoonright_\ell = \epsilon$ and $(\ell', n).\tau' \upharpoonright_\ell = (\ell', n).(\tau' \upharpoonright_\ell)$ if $\ell' \sqsubseteq \ell$; otherwise $(\ell', n).\tau' \upharpoonright_\ell = \tau' \upharpoonright_\ell$. Two traces τ_1, τ_2 are ℓ-equivalent, written $\tau_1 \approx_\ell \tau_2$, iff $\tau_1 \upharpoonright_\ell = \tau_2 \upharpoonright_\ell$.

2.2 Semantics

The operational semantics of SIMPL is standard and it is reported in Figure 10 in Appendix A.1. A state $(\mathcal{E}, m)$ is a pair of an environment $\mathcal{E} \in \mathit{Env}$ and a memory $m \in \mathit{Mem}$. A configuration $\mathcal{E} \vdash \langle P, m \rangle$ consists of an environment $\mathcal{E}$, a program P and a memory m. We write $\mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P', m' \rangle$ to denote that a configuration $\mathcal{E} \vdash \langle P, m \rangle$ evaluates in one step to configuration $\mathcal{E}' \vdash \langle P', m' \rangle$, producing an observation $\alpha \in \mathit{Obs}$. We write $\to^*$ or $\xrightarrow{\tau}^*$ to denote the reflexive and transitive closure of $\to$. We write $\mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\tau'}$ whenever the configuration is unimportant. We use ε to denote program termination.

2.3 Defining Secrecy

The goal of this subsection is to provide an attacker-centric definition of secrecy. The condition requires that the knowledge acquired by observing program outputs does not enable the attacker to learn sensitive information about the initial program state (inputs and memories). We assume the attacker knows the program code and has *perfect recall* of all the past observations. We first illustrate the security condition by an example, and then provide the formal definition.

Example 1. Let $P = \textbf{if } h \textbf{ then } \textbf{out}(\mathbf{L}, 1) \textbf{ else } \textbf{out}(\mathbf{L}, 2)$ be a SIMPL program and h a secret variable, i.e. $\Gamma(h) = \mathbf{H}$. Depending on the initial value of h, the program outputs either $\textbf{out}(\mathbf{L}, 1)$ or $\textbf{out}(\mathbf{L}, 2)$ on a channel of security level $\mathbf{L}$.

An attacker at security level $\mathbf{L}$ can reason about the initial value of h as follows: (i) Before seeing any output, the attacker considers any boolean value as possible for h, therefore the knowledge is $h \in \{\mathbf{tt}, \mathbf{ff}\}$. (ii) If the statement $\textbf{out}(\mathbf{L}, 1)$ is executed, the attacker can refine the knowledge to $h \in \{\mathbf{tt}\}$ and thus learn the initial value of h. (iii) Similarly, if the statement $\textbf{out}(\mathbf{L}, 2)$ is executed, the attacker learns that h was initially false. Hence, the program is insecure.

We now define the knowledge that an attacker at level ℓ acquires from observing a trace of a program P. We capture this by considering the set of initial states that the attacker considers possible based on their observations. Concretely, for a given initial state $(\mathcal{E}_0, m_0)$ and a program P, an initial state $(\mathcal{E}, m)$ is considered possible if $\mathcal{E} \approx_\ell \mathcal{E}_0$, $m \approx_\ell m_0$, and it matches the trace produced by $\mathcal{E}_0 \vdash \langle P, m_0 \rangle$. We define the attacker's knowledge in the standard way [4]:

Definition 1 (Knowledge). *The* knowledge set *for program P, initial state $(\mathcal{E}_0, m_0)$, security level ℓ and trace τ is given by $k(P, \mathcal{E}_0, m_0, \tau) = \{(\mathcal{E}, m) \mid \mathcal{E} \approx_\ell \mathcal{E}_0 \wedge m \approx_\ell m_0 \wedge (\exists P', \mathcal{E}', m', \tau'.\ \mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m' \rangle \wedge \tau \approx_\ell \tau')\}$.*

We focus on *progress-insensitive* security, which ignores information leaks through the observation of computation progress, e.g. program divergence [3]. To this end, we relax the requirement that the attacker learns nothing at each execution step, by allowing leaks that arise from observing the progress of computation. Concretely, we define progress knowledge as the set of initial states that the attacker considers possible based on the fact that *some* output event has occurred, independently of what the exact output value was.

Definition 2 (Progress Knowledge). *The* progress knowledge set *for program P, initial state $(\mathcal{E}_0, m_0)$, level ℓ, and trace τ is given by $k_P(P, \mathcal{E}_0, m_0, \tau) = \{(\mathcal{E}, m) \mid \mathcal{E} \approx_\ell \mathcal{E}_0 \wedge m \approx_\ell m_0 \wedge (\exists P', \mathcal{E}', m', \tau', \alpha \neq \epsilon.\ \mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m' \rangle \xrightarrow{\alpha}{}^{*} \wedge \alpha \restriction_\ell = \alpha \wedge \tau \approx_\ell \tau')\}$.*

We can now define a *progress-insensitive* secrecy by requiring that progress knowledge after observing a trace τ is the same as the knowledge obtained after observing the trace $\tau.\alpha$. Consequently, what the attacker learns from observing the exact output value is the same as what they learn from observing the computation progress, i.e. that some output event has occurred.

Definition 3 (Progress-insensitive Secrecy). *A program P satisfies* Progress-insensitive Secrecy *at level ℓ, written $Sec(\ell) \vDash P$, iff whenever $\mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\tau.\alpha}{}^{*} \mathcal{E}' \vdash \langle P', m' \rangle \wedge \alpha \restriction_\ell = \alpha \wedge \alpha \neq \epsilon$, we have $k_P(P, \mathcal{E}, m, \tau) = k(P, \mathcal{E}, m, \tau.\alpha)$. P satisfies* Progress-insensitive Secrecy*, written $Sec \models P$ iff $Sec(\ell) \vDash P$, for all ℓ.*

We can see that the program in Ex. 1 does not satisfy progress-insensitive secrecy at security level $\mathbf{L}$, as the progress knowledge of observing *some* output, i.e. either $\mathbf{out}(\mathbf{L}, 1)$ or $\mathbf{out}(\mathbf{L}, 2)$, is $h \in \{\mathbf{tt}, \mathbf{ff}\}$, while the knowledge of observing the *exact* output, e.g. $\mathbf{out}(\mathbf{L}, 1)$, is $h \in \{\mathbf{tt}\}$.

2.4 Security Conditions

Information-flow monitors can enforce progress-insensitive secrecy, thus preventing both implicit and explicit flows. Taint tracking, on the other hand, is an enforcement mechanism that only prevents explicit flows, otherwise ignores any control-flow dependencies [21]. In contrast to noninterference, security conditions for taint tracking [59,47] serve more as semantic criteria for evaluating soundness and precision of the underlying enforcement mechanism rather than providing an intuitive meaning of security. Driven by the same motivation, we propose a family of security conditions that allows exploring the space of enforcement mechanisms from taint tracking to information-flow control.

Our security conditions rely on the observational power of an attacker over the program code and executions. We model attackers with respect to their per-run view of the program code and extract the program *slice* that an attacker considers possible for any concrete execution. This allows to re-use the same condition as in Def. 3 for the program *slice* that the attacker can observe.

Concretely, a security condition for taint tracking can be modelled as secrecy with respect to an attacker that only observes explicit statements (input, output and assignment) extracted from any concrete execution of a program P. Similarly, (termination-insensitive) noninterference [3] corresponds secrecy for an attacker that has a whole view of P.

$$l_1 := \mathbf{tt}\ ;\ l_2 := \mathbf{tt}\ ; \tag{1}$$

$$\mathbf{if}\ h\ \mathbf{then}\ l_1 := \mathbf{ff}\ \mathbf{else\ skip} \tag{2}$$

$$\mathbf{if}\ l_1\ \mathbf{then}\ l_2 := \mathbf{ff}\ \mathbf{else\ skip} \tag{3}$$

$$\mathbf{out}(\mathbf{L}, l_2) \tag{4}$$

Fig. 3. Leaking through label upgrades

We will use the example in Figure 3 to illustrate the security conditions. Consider the program P with boolean variable h of level $\mathbf{H}$ and boolean variables l_1, l_2 of level $\mathbf{L}$. It can be seen that P outputs the initial value of variable h to an observer at security level $\mathbf{L}$ through a sequence of control flow decisions. In fact, the program does not satisfy the condition in Def. 3.

We introduce *extraction contexts* C as a *gadget* to model the observational power of an attacker over the program code. Extraction contexts provide a mechanism to leverage the operational semantics of the language and extract the program slice that an attacker observes for any given concrete execution.

$$C ::= [] \mid \mathbf{skip} \mid x := e \mid x \leftarrow \mathbf{in}(\ell) \mid \mathbf{out}(\ell, e) \mid C; C \mid \mathbf{if}\ e\ \mathbf{then}\ C\ \mathbf{else}\ C$$

Syntactically, extraction contexts are programs that may contain *holes* []. For our purposes, contexts will contain at most one hole that represents a placeholder for the program statements that are yet to be evaluated by the program execution at hand. We extend the operational semantics to transform contexts in order to extract programs for *weak secrecy* and *observable secrecy*.

Weak Secrecy Weak secrecy [59], a security condition for taint tracking, states that every sequence of explicit statements executed by any program run must be secure. We formalize weak secrecy as secrecy (cf. Def. 3) for the program, i.e. the sequence of explicit statements, extracted from any (possibly incomplete) execution of the original program. We achieve this by extending the configurations with extraction contexts. Here we discuss a few interesting rules as reported in Figure 4. The complete set of rules can be found in Figure 11 in Appendix A.2.

W-Assign
$$\frac{m(e) = n}{\mathcal{E} \vdash \langle x := e, m, C\rangle \rightarrow \mathcal{E} \vdash \langle \varepsilon, m[x \mapsto n], C[x := e]\rangle}$$

W-Out
$$\frac{m(e) = n}{\mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m, C\rangle \xrightarrow{[(\ell,n)]} \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{out}(\ell, e)]\rangle}$$

W-IfTrue
$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m, C\rangle \rightarrow \mathcal{E} \vdash \langle P_1, m, C\rangle}$$

W-Seq
$$\frac{\mathcal{E} \vdash \langle P_1, m, C\rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1', m', C'\rangle}{\mathcal{E} \vdash \langle P_1\ ;\ P_2, m, C\rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1'\ ;\ P_2, m', C'\rangle}$$

Fig. 4. Excerpt of Extraction Rules for Weak Secrecy

Each program execution starts with the empty context []. To extract explicit statements, we propagate assignment and output commands into the context, while conditionals are simply ignored (cf. the context remains unchanged). Sequential composition ensures that the sequence of explicit statements is propagated correctly. It can be shown that complete (terminated) executions contain no holes and incomplete executions contain exactly one hole.

We define weak secrecy in terms of secrecy for explicit statements extracted from any program execution. We write $C[\mathbf{skip}]$ to denote the result of replacing the hole with command **skip** in a context C. Otherwise, if the context contains no hole, we have $C[\mathbf{skip}] = C$. This is needed because the security condition is defined for any execution, including complete and incomplete executions.

Definition 4 (Weak secrecy). *A program P satisfies* weak secrecy *for initial state $(\mathcal{E}, m)$, written $WS \vDash_{\mathcal{E},m} P$, iff whenever $\mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow{\tau}^{*} \mathcal{E}' \vdash \langle P', m', C\rangle$, we have $Sec \models C[\boldsymbol{skip}]$. A program P satisfies* weak secrecy*, written $WS \vDash P$, iff $WS \vDash_{\mathcal{E},m} P$ for all states $(\mathcal{E}, m)$.*

Consider the program from Ex. 3 and an initial state $(\mathcal{E}_0, m_0)$. Depending on whether $m_0(h) = \mathbf{tt}$ and $m_0(h) = \mathbf{ff}$, we extract program (5) or program (6), respectively, shown in Figure 5.

$$l_1 := \mathbf{tt}\ ;\ l_2 := \mathbf{tt}\ ;\ l_1 := \mathbf{ff}\ ;\ \mathbf{skip}\ ;\ \mathbf{out}(\mathbf{L}, l_2) \qquad (5)$$
$$l_1 := \mathbf{tt}\ ;\ l_2 := \mathbf{tt}\ ;\ \mathbf{skip}\ ;\ l_2 := \mathbf{ff}\ ;\ \mathbf{out}(\mathbf{L}, l_2) \qquad (6)$$

Fig. 5. Extracted programs

We can see that none of the programs contains variable h, hence they both satisfy secrecy (Def. 3). As a result, the original program P satisfies weak secrecy.

Observable Secrecy We now present a novel security condition, dubbed *observable secrecy*, that captures the intuition of *observable* implicit flows. Observable implicit flows are implicit flows that arise whenever a variable is modified in the high branch that is currently executed by the program, and later it is output to the attacker. Preventing observable implicit flows is of interest for purely

O-Skip

$$\frac{}{\mathcal{E} \vdash \langle \mathbf{skip}, m, C\rangle \to \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{skip}]\rangle}$$

O-In

$$\frac{\mathcal{E}' = \mathcal{E}[\ell \mapsto n \mapsto \mathcal{E}(\ell)(n+1)] \qquad m' = m[x \mapsto \mathcal{E}(\ell)(0)]}{\mathcal{E} \vdash \langle x \leftarrow \mathbf{in}(\ell), m, C\rangle \to \mathcal{E}' \vdash \langle \varepsilon, m', C[x \leftarrow \mathbf{in}(\ell)]\rangle}$$

O-Assign

$$\frac{m(e) = n}{\mathcal{E} \vdash \langle x := e, m, C\rangle \to \mathcal{E} \vdash \langle \varepsilon, m[x \mapsto n], C[x := e]\rangle}$$

O-Seq

$$\frac{\mathcal{E} \vdash \langle P_1, m, C\rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1', m', C'\rangle}{\mathcal{E} \vdash \langle P_1 \ ; \ P_2, m, C\rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1' \ ; \ P_2, m', C'\rangle}$$

O-Out

$$\frac{m(e) = n}{\mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m, C\rangle \xrightarrow{[(\ell,n)]} \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{out}(\ell, e)]\rangle}$$

O-WhileFalse

$$\frac{m(e) = \mathbf{ff}}{\mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m, C\rangle \to \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{skip}]\rangle}$$

O-WhileTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m, C\rangle \to \mathcal{E} \vdash \langle P \ ; \ \mathbf{while}\ e\ \mathbf{do}\ P, m, C[\mathbf{if}\ e\ \mathbf{then}\ []\ \mathbf{else\ skip}]\rangle}$$

O-IfTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m, C\rangle \to \mathcal{E} \vdash \langle P_1, m, C[\mathbf{if}\ e\ \mathbf{then}\ []\ \mathbf{else\ skip}]\rangle}$$

O-SeqEmpty

$$\frac{}{\mathcal{E} \vdash \langle \varepsilon \ ; \ P_2, m, C\rangle \to \mathcal{E} \vdash \langle P_2, m, C \ ; \ []\rangle}$$

O-IfFalse

$$\frac{m(e) = \mathbf{ff}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m, C\rangle \to \mathcal{E} \vdash \langle P_2, m, C[\mathbf{if}\ e\ \mathbf{then\ skip\ else}\ []]\rangle}$$

O-IfTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m, C\rangle \to \mathcal{E} \vdash \langle P_1, m, C[\mathbf{if}\ e\ \mathbf{then}\ []\ \mathbf{else\ skip}]\rangle}$$

O-SeqEmpty

$$\frac{}{\mathcal{E} \vdash \langle \varepsilon \ ; \ P_2, m, C\rangle \to \mathcal{E} \vdash \langle P_2, m, C \ ; \ []\rangle}$$

Fig. 6. Extraction Rules for Observable Secrecy

dynamic mechanisms as it provides higher security compared to weak secrecy, yet allowing for dynamic monitors that are more permissive than monitors for noninterference. Permissiveness, however, comes at the price of ignoring hidden implicit flows. The following program, where h has security level $\mathbf{H}$, contains an observable implicit flow whenever $m_0(h) = \mathbf{tt}$, otherwise the flow is hidden.

$$l := \mathbf{ff} \ ; \mathbf{if}\ h\ \mathbf{then}\ \{l := \mathbf{tt}\}\ \mathbf{else}\ \{\mathbf{skip}\} \ ; \mathbf{out}(\mathbf{L}, l)$$

The security condition considers an attacker that only observes the instructions (both control-flow and explicit statements) executed by the concrete program execution, otherwise it ignores (i.e. replaces with **skip**) any instruction occurring in the untaken branches. To capture these flows, we extend the small-step operational semantics to extract the program code observable by this attacker, as shown in Figure 6.

The rules for assignment, input, output and sequential composition are the same as for weak secrecy. Rules for conditionals propagate the *observable* conditional into the context C to keep track of the executed branch and replace the untaken branch with **skip**. The new hole [] ensures that the commands under the executed branch are properly modified by the new context. We unfold loop statements into conditionals and handle them similarly. Sequential composition ensures that the sequence of observable statements is propagated correctly. When rule O-SeqEmpty is applied, the context C does not contain any holes, hence a new hole is introduced to properly handle the remaining command P_2.

Definition 5 (Observable secrecy). *A program P satisfies* observable secrecy for initial state $(\mathcal{E}, m)$*, written* $OS \vDash_{\mathcal{E},m} P$*, iff whenever* $\mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m', C\rangle$*, we have* $Sec \models C[\boldsymbol{skip}]$*. A program P satisfies* observable secrecy*, written* $OS \vDash P$*, iff* $OS \vDash_{\mathcal{E},m} P$ *for all states* $(\mathcal{E}, m)$*.*

For the above example, the operational semantics rules for observable secrecy yield the programs:

$$l := \mathbf{ff}\,;\ \mathbf{if}\ h\ \mathbf{then}\ \{l := \mathbf{tt}\}\ \mathbf{else}\ \{\mathbf{skip}\}\ ;\ \mathbf{out}(\mathbf{L}, l)$$
$$l := \mathbf{ff}\,;\ \mathbf{if}\ h\ \mathbf{then}\ \{\mathbf{skip}\}\ \mathbf{else}\ \{\mathbf{skip}\}\ ;\ \mathbf{out}(\mathbf{L}, l)$$

The first program does not satisfy secrecy (Def. 3), while the second program does. Therefore the original program does not satisfy observable secrecy.

Full Secrecy Full secrecy is a security condition that models secrecy with respect to an attacker that has a complete knowledge of program code and therefore can learn information through explicit and (observable or hidden) implicit flows. This corresponds to progress-insensitive noninterference (Def. 3).

Definition 6 (Full secrecy). *A program P satisfies* full secrecy for initial state $(\mathcal{E}, m)$*, written* $FS \vDash_{\mathcal{E},m} P$*, iff whenever* $\mathcal{E} \vdash \langle P, m\rangle \xrightarrow{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m'\rangle$*, we have* $Sec \models P$*. A program P satisfies* full secrecy*, written* $FS \vDash P$*, iff* $FS \vDash_{\mathcal{E},m} P$ *for all states* $(\mathcal{E}, m)$*.*

3 Enforcement Framework

We employ variants of flow-sensitive dynamic monitors (trackers) to enforce the security conditions presented in the last section. Compared to existing work (cf. Sect. 6), we use semantic security conditions, weak secrecy and observable secrecy, to justify soundness of *weak* tracking and *observable* tracking mechanisms.

Figure 7 presents the instrumented semantics which is parametric on the security labels, transfer functions and constraints. By instantiating each of the parameters (Table. 1), we show how the semantics implements sound dynamic trackers for weak secrecy (Thrm. 1), observable secrecy (Thrm. 2) and full secrecy (Thrm. 3). All proofs are reported in Appendix B.

The instrumented semantics assumes a bounded lattice $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$ and an initial security environment Γ, as defined in Sect. 2.1. We use a *program counter* stack of security levels pc to keep track of the security context, i.e. the security level of conditional and loop expressions, at a given execution point. We write $\ell :: pc$ to denote a stack of labels, where the label ℓ is its top element. Abusing notation, we also write pc to represent the upper bound on the security levels of the stack elements. The monitored semantics introduces the special instruction **end** to remember the join points in the control flow and update the pc stack accordingly. Instrumented configurations $\Gamma, pc, \mathcal{E} \vdash \langle P, m\rangle$ extend original configurations with the security environment Γ and security context stack pc. We write $\Gamma, pc, \mathcal{E} \vdash \langle c, m\rangle \xrightarrow{\alpha}\!\!\!\!\!\twoheadrightarrow \Gamma', pc', \mathcal{E}' \vdash \langle c', m'\rangle$ to denote that an instrumented configuration $\Gamma, pc, \mathcal{E} \vdash \langle c, m\rangle$ evaluates in one step to instrumented configuration $\Gamma', pc', \mathcal{E}' \vdash \langle c', m'\rangle$, producing observations $\alpha \in Obs$. We write $\twoheadrightarrow^*$ or

S-Skip

$$\frac{}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{skip}, m\rangle \twoheadrightarrow \Gamma, pc, \mathcal{E} \vdash \langle \varepsilon, m\rangle}$$

S-In-F

$$\frac{\phi_{inF}}{\Gamma, pc, \mathcal{E} \vdash \langle x \leftarrow \mathbf{in}(\ell), m\rangle \twoheadrightarrow \lightning}$$

S-Out

$$\frac{m(e) = n \qquad \phi_{outT}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m\rangle \overset{[(\ell,n)]}{\twoheadrightarrow} \Gamma, pc, \mathcal{E} \vdash \langle \varepsilon, m\rangle}$$

S-WhileFalse

$$\frac{m(e) = \mathbf{tt} \qquad \phi_{wh}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m\rangle \twoheadrightarrow \Gamma, pc', \mathcal{E} \vdash \langle \mathbf{end}, m\rangle}$$

S-In

$$\frac{\mathcal{E}' = \mathcal{E}[\ell \mapsto n \mapsto \mathcal{E}(\ell)(n+1)] \qquad m' = m[x \mapsto \mathcal{E}(\ell)(0)] \qquad \Gamma' = \Gamma[x \mapsto \ell \sqcup pc] \qquad \phi_{inT}}{\Gamma, pc, \mathcal{E} \vdash \langle x \leftarrow \mathbf{in}(\ell), m\rangle \twoheadrightarrow \Gamma', pc, \mathcal{E}' \vdash \langle \varepsilon, m'\rangle}$$

S-IfFalse

$$\frac{m(e) = \mathbf{ff} \qquad \phi_{if}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m\rangle \twoheadrightarrow \Gamma, pc', \mathcal{E} \vdash \langle P_2\ ;\ \mathbf{end}, m\rangle}$$

S-Assign-F

$$\frac{\phi_{asgF}}{\Gamma, pc, \mathcal{E} \vdash \langle x := e, m\rangle \twoheadrightarrow \lightning}$$

S-IfTrue

$$\frac{m(e) = \mathbf{tt} \qquad \phi_{if}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m\rangle \twoheadrightarrow \Gamma, pc', \mathcal{E} \vdash \langle P_1\ ;\ \mathbf{end}, m\rangle}$$

S-Out-F

$$\frac{\phi_{outF}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m\rangle \twoheadrightarrow \lightning}$$

S-SeqEmpty

$$\frac{}{\Gamma, pc, \mathcal{E} \vdash \langle \varepsilon\ ;\ P_2, m\rangle \twoheadrightarrow \Gamma, pc, \mathcal{E} \vdash \langle P_2, m\rangle}$$

S-End

$$\frac{\phi_{End}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{end}, m\rangle \twoheadrightarrow \Gamma, pc', \mathcal{E} \vdash \langle \varepsilon, m\rangle}$$

S-WhileTrue

$$\frac{m(e) = \mathbf{tt} \qquad \phi_{wh}}{\Gamma, pc, \mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m\rangle \twoheadrightarrow \Gamma, pc', \mathcal{E} \vdash \langle P\ ;\ \mathbf{end}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P, m\rangle}$$

S-Assign

$$\frac{m(e) = n \qquad \Gamma' = \Gamma[x \mapsto pc \sqcup \Gamma(e)] \qquad \phi_{asgT}}{\Gamma, pc, \mathcal{E} \vdash \langle x := e, m\rangle \twoheadrightarrow \Gamma', pc, \mathcal{E} \vdash \langle \varepsilon, m[x \mapsto n]\rangle}$$

S-Seq

$$\frac{\Gamma, pc, \mathcal{E} \vdash \langle P_1, m\rangle \overset{\alpha}{\twoheadrightarrow} \Gamma', pc', \mathcal{E}' \vdash \langle P_1', m'\rangle}{\Gamma, pc, \mathcal{E} \vdash \langle P_1\ ;\ P_2, m\rangle \overset{\alpha}{\twoheadrightarrow} \Gamma', pc', \mathcal{E}' \vdash \langle P_1'\ ;\ P_2, m'\rangle}$$

Fig. 7. Instrumented Semantics

$\overset{\tau}{\twoheadrightarrow}^*$ to denote the reflexive and transitive closure of $\overset{\alpha}{\twoheadrightarrow}$. We write $\Gamma(e)$ for $\sqcup_{x \in Vars(e)} \Gamma(x)$ and $\lightning$ for abnormal termination.

In what follows, we use the constraints in Table 1 to instantiate the rules in Figure 7, and present a family of dynamic monitors for weak tracking (known as taint tracking), observable tracking, and full tracking (known as No-Sensitive Upgrade [5]). The monitors implement the *failstop* strategy and terminate the program abnormally (cf. rules for $\lightning$) whenever a potentially insecure statement is executed. Note that abnormal termination does not produce any observable event and it is treated as a progress channel, similarly to nontermination. We write $\mathcal{I} \vdash_{\mathcal{E},m} P$ for an execution of a monitored program P from initial state $(\mathcal{E}, m)$, initial security environment Γ and initial stack $\bot$, where $\mathcal{I} \in \{WS, OS, FS\}$.

Monitored executions may change the semantics of the original program by collapsing insecure executions into abnormal termination. To account for the monitored semantics, we instantiate the security conditions from Section 2.4 with the semantics of instrumented executions and, abusing notation, write $\mathcal{I} \models_{\mathcal{E},m} P$ to refer to an execution of P under the instrumented semantics. We then show that any program executed under an instrumented execution, i.e, $\mathcal{I} \vdash_{\mathcal{E},m} P$, satisfies the security condition, i.e., $\mathcal{I} \models_{\mathcal{E},m} P$.

Weak Tracking Weak tracking is a dynamic mechanism that prevents explicit flows from sources of higher security levels to sinks of lower security levels. Weak tracking allows leaks through implicit flows. The second column in Table 1 gives the set of constraints that a typical taint analysis would implement for our language.

Since the analysis ignores all implicit flows, the *pc* stack is redundant and we never update it during the monitor execution. For the same reason, we apply no side conditions to the rules for conditionals and loops. Rule S-Assign propagates the security level of the expression on the right-hand side to the variable on the left-hand side to track potential explicit flows, while rule S-Assign-F never applies. Rule S-Out ensures that only direct flows from lower levels affect a given output level. If the constraint is not satisfied, the program terminates abnormally (cf. S-Out-F).

Rule	Weak	Observable	Full
ϕ_{asgT}	$\mathbf{tt}$	$\mathbf{tt}$	$pc \sqsubseteq \Gamma(x)$
ϕ_{asgF}	$\mathbf{ff}$	$\mathbf{ff}$	$pc \not\sqsubseteq \Gamma(x)$
ϕ_{outT}	$\Gamma(e) \sqsubseteq \ell$	$\Gamma(e) \sqsubseteq pc \sqcup \ell$	$\Gamma(e) \sqsubseteq pc \sqcup \ell$
ϕ_{outF}	$\Gamma(e) \not\sqsubseteq \ell$	$\Gamma(e) \not\sqsubseteq pc \sqcup \ell$	$\Gamma(e) \not\sqsubseteq pc \sqcup \ell$
ϕ_{inT}	$\mathbf{tt}$	$pc \sqsubseteq \ell$	$pc \sqsubseteq \ell$
ϕ_{inF}	$\mathbf{ff}$	$pc \not\sqsubseteq \ell$	$pc \not\sqsubseteq \ell$
ϕ_{end}	$\mathbf{tt}$	$pc = \ell :: pc'$	$pc = \ell :: pc'$
ϕ_{if} / ϕ_{wh}	$\mathbf{tt}$	$\ell' = pc \sqcup \Gamma(e)$ $pc' = \ell' :: pc$	$\ell' = pc \sqcup \Gamma(e)$ $pc' = \ell' :: pc$

Table 1. Constraints for Monitors in Fig. 7

To illustrate the weak tracking monitor, consider the program from Ex. 3. Initially, the security environment Γ assigns the label $\mathbf{L}$ to variables l_1 and l_2, and the label $\mathbf{H}$ to variable h. After the execution of line (1), the security environment Γ' does not change since $pc = \mathbf{L}$ and, $\Gamma(n) = \mathbf{L}$ for all $n \in \mathit{Val}$, therefore $\Gamma'(l_1) = \Gamma'(l_2) = \mathbf{L} \sqcup \Gamma(\mathbf{ff}) = \mathbf{L}$ (cf. rule S-Assign). Moreover, the lines (2) and (3) do not modify Γ' (cf. rules S-IfTrue and S-IfFalse). Finally, the output in line (4) is allowed since $\Gamma(l_2) = \mathbf{L} \sqsubseteq \mathbf{L}$ (cf. rule S-Out). In fact, the program satisfies weak secrecy (Def. 4), and it is accepted by weak tracking.

We show that any program that is executed under the weak tracking monitor, i.e. $\mathcal{I} = WS$, satisfies weak secrecy.

Theorem 1. $WS \vdash_{\mathcal{E},m} P \Rightarrow WS \vDash_{\mathcal{E},m} P$

Observable Tracking Observable tracking is a dynamic security mechanism that accounts for explicit flows and observable implicit flows. Observable implicit flows occur whenever a low security variable that is updated in a high security context is later output to a low security channel. The condition justifies the security of a program with respect to an attacker that only knows the control-flow path of the current execution. Observable tracking has the appealing property of only propagating the security label of variables in a concrete program execution, without analyzing variables modified in the untaken branches. This is remarkable as it sidesteps the need for convoluted static analysis otherwise required for languages with dynamic features such as reflection. Moreover, as we discuss later, observable tracking is more permissive than existing enforcement mechanisms such as NSU [5] or Permissive Upgrade [6]. Permissiveness is achieved at the expense of enforcing a different security condition, i.e. observable secrecy, instead of full secrecy. For trusted code, observable secrecy might be sufficient to determine unintentional security bugs. Otherwise, for malicious code, we present a transformation (Sect. 4) that enables observable tracking to enforce full secrecy, yet being more permissive than full tracking.

The instrumented semantics for observable tracking (cf. third column in Table 1) strengthens the constraints for weak tracking by: (i) introducing the *pc* stack to properly track changes of security labels for variables updated in a high context; (ii) disallowing input from low security channels in a high context; (iii) and constraining the output on a low channel by disallowing low expressions that depend on a high context.

Consider again the program in Ex. 3 under the instrumented semantics for observable tracking. After executing the assignments in (1), the variables l_1 and l_2 have security level **L**. If h is **tt**, the variable l_1 has security level **H** after the first conditional in (2) (cf. S-IfTrue rule). As a result, the guard of the second conditional in (3) is false, and we execute the **else** branch. The security level of the variable l_2 remains **L**, therefore the output on the **L** channel in (4) is allowed (cf. S-Out rule). Otherwise, if h is **ff**, then the **else** branch is executed and l_1 has security level **L**. The second conditional does not change the security level of l_2, although the **then** branch is executed. In fact, the guard only depends on **L** variables, i.e. l_1, hence security level of l_2 remains **L** and the subsequent output is allowed. The program, in fact, satisfies observable secrecy.

We prove that any program that is executed under the observable tracking monitor, i.e. $\mathcal{I} = OS$, satisfies observable secrecy.

Theorem 2. $OS \vdash_{\mathcal{E},m} P \Rightarrow OS \vDash_{\mathcal{E},m} P$

Full Tracking Full tracking, best known as No-Sensitive Upgrade [60,5], prevents both explicit and (observable or hidden) implicit flows from sources of higher security levels to sinks of lower security levels. This is achieved by disallowing changes of variables' security labels in high contexts (as opposed to the strategy followed by observable tracking). While sound for full secrecy, this strategy incorrectly terminates any program that updates a low security variable in a high security context, even if that variable is never output to low channel. This is unfortunate as it rejects secure programs that only use sensitive data for internal computations without ever sending them on low channels.

The semantics for full tracking adds additional constraints to the rules for observable tracking (cf. fourth column in Table 1). In particular, rule S-Assign only allows low assignments in low security contexts, i.e. whenever $pc \sqsubseteq \Gamma(x)$.

Consider again the program in Ex. 3 and the semantics for full tracking. As before, initially $\Gamma(l_1) = \Gamma(l_2) = \mathbf{L}$, and $\Gamma(h) = \mathbf{H}$. If the value of h is true, the **then** branch of the first conditional is executed, and the program is stopped because of a low assignment in a high context. This is a sound behavior of full tracking as the original program does not satisfy full secrecy. Unfortunately, full tracking will also stop any secure programs that contain the conditional statement in (2). For example, if we replace the output statement in (4) with $\mathbf{out}(\mathbf{L}, 1)$ or $\mathbf{out}(\mathbf{H}, l_2)$, the resulting program clearly satisfies full secrecy. However, whenever h is true, full tracking will incorrectly stop the program.

We show that any program that is executed under the full tracking monitor, i.e. $\mathcal{I} = FS$, satisfies full secrecy.

Theorem 3. $FS \vdash_{\mathcal{E},m} P \Rightarrow FS \vDash_{\mathcal{E},m} P$

	PROGRAM $\Gamma(h) = \mathbf{H}$, $\Gamma(l) = \Gamma(k) = \mathbf{L}$ and $h = \mathbf{tt}$	WEAK	FULL	PU	OT
P_0	$l := \mathbf{tt}$; **if** h **then** $\{l := h\}$; $\mathbf{out}(\mathbf{L}, l)$	−	−	−	−
P_1	**if** h **then** $l := \mathbf{tt}$	+	−	+	+
P_2	**if** h **then** $l := \mathbf{tt}$; **if** l **then skip**	+	−	−	+
P_3	$l := \mathbf{tt}$; $k := \mathbf{tt}$; **if** h **then** $\{l := \mathbf{ff}\}$; **if** l **then** $\{k := \mathbf{ff}\}$; $\mathbf{out}(\mathbf{L}, 1)$	+	−	−	+
P_4	**if** h **then** $\mathbf{out}(\mathbf{L}, 1)$ **else** $\mathbf{out}(\mathbf{L}, 1)$	+	−	−	−
P_5	$l := \mathbf{tt}$; $k := \mathbf{tt}$; **if** h **then** $\{l := \mathbf{ff}\}$; **if** l **then** $\{k := \mathbf{ff}\}$; $\mathbf{out}(\mathbf{L}, k)$	+	✗	✗	+

Table 2. Permissiveness

4 Staged Information-Flow Control

Two main factors hinder the adoption of dynamic information-flow control in practice: *challenging implementation* and *permissiveness*. To properly update the program counter stack at runtime, observable and full tracking require the knowledge of the program's control-flow graph. This requirement is unrealistic for unstructured, heavily optimized or obfuscated code, such as the code delivered to end users (cf. Sect. 1). In contrast, weak tracking disregards the control-flow graph and only considers explicit statements. As a result, the enforcement is more permissive and easier to implement.

In Appendix C, we present a staged analysis that first applies lightweight program transformations to convert implicit flows into explicit flows, thus delegating the task of enforcing observable and full secrecy to a weak tracker. Concretely, we inline the program counter stack into the source code in a semantics-preserving manner by introducing fake dependencies that cause a weak tracker to capture potential observable and/or hidden implicit flows. The transformation is completely transparent to the underlying security policy, which makes it suitable for the scenarios envisioned in Sect. 1. Full proofs are reported in Appendix D.

Soundness vs Permissiveness We use the examples in Table 2 to illustrate soundness and permissiveness for existing dynamic trackers. Except for the program P_5, all programs are secure for full secrecy. We summarize the relations between the security conditions (solid ovals) and enforcement mechanisms (dashed ovals) in Figure 4. The security conditions are incomparable, as shown by the programs P_0, P_4 and P_5 from Table 2. Moreover, there is a strict inclusion between the set of secure programs accepted by the trackers (cf. Table 2).

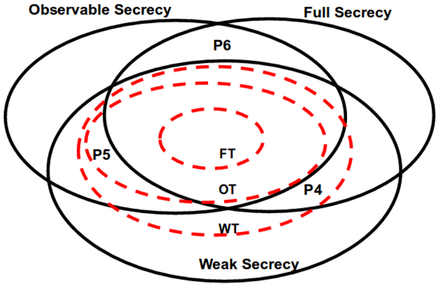

Fig. 8. Soundness vs Permissiveness

Theorem 4. $FT \vdash_{\mathcal{E},m} P \Rightarrow OT \vdash_{\mathcal{E},m} P \Rightarrow WT \vdash_{\mathcal{E},m} P$

Proof. The proof follows from the constraints in Table 1. The constraints for full tracking are stronger than the constraints for observable tracking, which are stronger than the constraints for weak tracking. Hence, the implications follow immediately.

Table 2 illustrates permissiveness for the state-of-the-art purely dynamic trackers. All trackers account for explicit flows, however, as illustrated by program P_0, they can be imprecise (cf. “$-$”) due to approximation. P_1 will be rejected by full tracking, i.e NSU [5], while program P_2 will be rejected by Permissive Upgrade [6], although none of them performs any outputs. P_3 encodes the value of the high boolean variable h into the final value of variable k through hidden implicit flows, however, k is never output. Observable tracking (column 6 and 7) correctly accepts the program, thus decreasing the number of false positives that the other trackers would otherwise report. P_0 and P_4 will be rejected by most trackers due to over-approximation. Arguably, program patterns like P_0 and P_4 are unlikely to be used, and, for trusted code, they can be fixed, e.g. by code transformations.

These considerations make a good case for using observable tracking as a permissive purely dynamic mechanism for security testing. However, programs may still leak through hidden implicit flows. The insecure program P_5 will be correctly rejected by NSU and Permissive Upgrade (cf. “✗”) and, it will be correctly accepted by observable tracking.

5 Implementation and Evaluation

Implementation Our tool is a prototype built on top of the *Soot* framework [56] and it uses an intermediate bytecode language, *Jimple* [56], to implement the static transformations presented in Sect. 3. We provide a description of Jimple in Appendix E.4 and discuss advanced language features in Appendix E.We implemented the code transformation for Android applications. The instrumented applications are then run using TaintDroid [24]. The code of the implementation is available online [14]. Overall, the implementation of static transformations proved to be straight-forward, due to the use of Jimple as an intermediate language and the modularity of the transformations. This indicates that this approach is indeed lightweight compared to elaborate information-flow trackers.

Use Case: Pedometer To evaluate our approach, we apply the presented implementation to an open-source step counting application [41] from the popular F-Droid repository. By default, the application performs no network output. To check if illegal flows are properly detected, we add network communication in a number of scenarios. We give condensed forms of these examples in this section to abstract from Android-specific issues regarding sensor queries; we refer the reader to the implementation's source code for the full examples [14].

Usage statistics: The step counting application may want to report usage information to the developer. However, a user may not want the actual step count to be reported to the developer. By tracking observable implicit flows, reporting usage information in a low context does not generate a false positive. However, disclosing the actual step count or reporting that the app was used on certain day in a high context will yield an error.

Declassifying average pace: The application may additionally send the average pace to a server to provide comparisons with other users. However, the actual step

```
$\textbf{if}\ (\mathit{stepSensor.newStep}() == \textbf{true})$
    $\textbf{then}\ \mathit{steps} := \mathit{steps} + 1\ \textbf{else skip}$
$\textbf{out}(L, \text{“App used on ”} + (\textbf{new}\ \mathit{Date}()))$
```

Fig. 9. Step counter example

count should still not be disclosed.
We implement a where-style declassification policy as described in Appendix E.

Location information: To show the user more detailed information, we also extended the application with rudimentary location tracking to allow for displaying information such as the number of steps per city. As location information is sensitive, our transformation ensures that nothing about the user's coordinates is leaked through explicit or observable implicit flows. We then modified the program to leak location information through hidden implicit flows as in Ex. 3. Again, our cross-copying transformation ensured that such leaks are prevented.

Use Case: JSFlow Existing information-flow tools, such as JSFlow [30], can be easily modified to enforce observable secrecy instead of noninterference. For the latest release of JSFlow, version 1.1, it was sufficient to comment out as few as 4 lines of code to change to enforcing observable secrecy.

Work on value sensitivity in the context of JSFlow [31] points out precision issues due to the No-Sensitive Upgrade policy, as in examples like ($x := 1$; $\textbf{if}\ h\ \textbf{then}\ x := 2\ \textbf{else}\ \textbf{skip}$; $\textbf{out}(\mathbf{L}, 1)$). A standard information-flow monitor such as JSFlow would stop this program to avoid upgrading the label of x in a secret context, even though x is never output later in the program. Modifying JSFlow to enforce observable secrecy however accepts the program.

6 Related Work

Referring to the surveys on language-based information-flow security [44] and taint tracking [49], we only discuss the most closely related work.

Information-Flow Policies Contrasting noninterference [28], Volpano [59] introduces weak secrecy, a security condition for taint tracking. Schoepe et al. generalize weak secrecy by explicit secrecy [47] and enforce it by faceted values [48]. Our work explores observable secrecy as the middle ground. Similarly to weak secrecy and noninterference, observable secrecy is not a trace property.

Several authors study knowledge-based conditions [4,3,9,10]. We explore the attacker's view of program code to discriminate polices, relating in particular to the *forgetful* attackers by Askarov and Chong [2], though the exact relation is subject to further investigation. While implicit flows in the wild are important [42,33], they can also appear in trusted code [35,34]. By tracking explicit and observable implicit flows, we raise the security bar wrt. taint tracking.

Staged Analysis Our work takes inspiration from Beringer [15], who provides formal arguments of using taint tracking to enforce noninterference policies. Beringer also leverages the cross copying technique to consider hidden implicit flows. By contrast, we justify soundness of the enforcement mechanism in terms of semantic conditions like weak secrecy with respect to *uninstrumented* semantics. On the other hand, Beringer introduces a notion of *path tracking* to account for termination-sensitive noninterference, and supports the theory (for an imperative language *without* I/O) by a formalization in Coq. Our work distinguishes between malicious and trusted code, providing security conditions and enforcement mechanisms for both settings (including a prototype implementation).

Rifle [55] treats implicit flows by cross-copying program instrumentation and taint tracking, with separate taint registers for explicit and implicit flows. The focus is on efficiency, as soundness is only justified informally. Like Beringer's, our work gives formal and practical evidence for the usefulness of Rifle's ideas.

Other works leverage the cross-copying technique to enforce noninterference policies. Le Guernic [36] uses cross-copying in a hybrid monitor for noninterference, and refers to observable and hidden implicit flows as implicit and explicit indirect flows, respectively. Chugh et al. [19] present a hybrid approach to handling JavaScript code. Their approach first computes statically a dynamic residual, which is checked at runtime in a second stage. For trusted code, Kang et al. [34] study targeted (called *culprit*) implicit flows. Bao et at. [11] identify *strict* control dependences and evaluate their effectiveness for taint tracking empirically. These works illuminate the benefits of observable implicit flows.

Dynamic Enforcement and Inlining Fenton [26] studies purely dynamic information-flow monitors. Austin and Flanagan [5] leverage No-Sensitive Upgrade [60] to enforce noninterference for JavaScript and propose Permissive Upgrade [6] to improve precision. We show that NSU can be too restrictive, and propose solutions to improve precision for malicious and trusted code. Chudnov and Naumann [18] and Magazinius et al. [37] propose information-flow monitor inlining, integrating the NSU strategy into program's code. Bielova and Rezk [17] survey recent work in (information-flow) monitor inlining. Our transformations can be seen as lightweight inlining of dynamic monitors, for (observable and/or hidden) implicit flows. Russo and Sabelfeld [43] discuss trade-offs between static and dynamic flow-sensitive analysis. We leverage their flow-sensitive monitor.

Secure multi-execution [22] and faceted values [7] enforce noninterference: programs are executed as many times as there are security levels, with outputs at each level computed by the respective runs. Barthe et al. [12] study program transformations to implement secure multi-execution. These techniques are secure by construction and provide high precision. However, they require synchronization between computations at different security levels, and face challenges for languages with side-effects and I/O. Also, they may modify the semantics and introduce crashes, thus making it difficult to detect attacks. By contrast, we focus on failstop monitoring, trading full permissiveness to avoids such pitfalls.

Static and Hybrid Enforcement Volpano et al. [58] formalize the soundness of Dennings' static analysis [21] with respect to noninterference by a security type system, extended by further work with advanced features [44]. Hunt and Sands [32] present flow-sensitive security types. Our work leverages dynamic analysis to enforce similar policies. Other analysis for information flow include program logics [10,13], model checking [8,23], abstract interpretations [27] and theorem proving [20,40]. While more precise than security type systems, these approaches may face several challenges with scalability.

Hybrid enforcement combines static and dynamic analysis. Le Guernic [36] proposes hybrid flow-sensitive mechanisms supporting for sequential and concurrent languages. Venkatakrishnan et al [57] present a hybrid monitor for a language with procedures and show that it enforces noninterference. Shroff et al. [50] present a monitor with dynamic dependency analysis for a language with heap. Tripp et al. [54] study hybrid security for JavaScript code by combining static analysis and dynamic partial evaluation. Moore and Chong [39] propose two optimizations of hybrid monitors for efficiency: selective tracking of variable security levels and memory abstractions for languages with dynamic memory. Hybrid approaches use static analysis to approximate computational effects for program paths that are not visited by a given execution. This can be challenging for languages with complex features, e.g. reflection, and unstructured control flow. We strike the balance by performing static analysis for implicit flows (basically boolean expressions) and delegating the resolution of complex features to a dynamic taint tracker.

Mobile App Security There exists a large body of works on information-flow analysis in the mobile app domain. The majority of these analysis only accounts for explicit flows. This is due to the presence of complex language features and highly dynamic lifecycles, however, for potentially malicious and trusted code, implicit flows are important to address. Our proposal in Figure 1 enables existing work to provide stronger guarantees in a flexible manner. TaintDroid [24] is a dynamic taint tracker developed to capture privacy violations in Android apps. We use TaintDroid as dynamic component in our implementation. Most static analysis works certify security with respect to weak secrecy [1,29]. Despite the great progress in improving precision, the false positive rate remains high [29].

Ernst et al. [25] propose collaborative verification of information-flow requirements for a high-integrity app store. Developers and the app store collaborate to reduce the overall verification cost. Concretely, developers provide the source code with information-flow specifications (security types), while the app store verifies their correctness. Our model is complementary and, by contrast, user-centric, allowing for more flexible policies and reducing the developers' burden.

7 Conclusion

We have presented a framework of information-flow trackers, allowing us to relate a range of enforcement from taint tracking to information-flow control. We have explored the middle ground by distinguishing malicious and trusted code and considering trade-offs between soundness and permissiveness. We have deployed the framework in a staged fashion by combining lightweight static analysis with dynamic taint tracking, enabling us to envision a secure app store architecture. We have experimented with the approach by a prototype implementation.

Future work includes dynamic security policies and case studies from the F-Droid repository. While the current framework allows for parametric policies on users' side, we conjecture that the static transformations, being transparent to the underlying policy, can be extended to handle rich dynamic policies.

Acknowledgments This work was partly funded by the European Community under the ProSecuToR project and the Swedish research agency VR.

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI (2014)
2. Askarov, A., Chong, S.: Learning is change in knowledge: Knowledge-based security for dynamic policies. In: CSF (2012)
3. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: ESORICS (2008)

4. Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: S&P (2007)
5. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. SIGPLAN Not. (2009)
6. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: PLAS (2010)
7. Austin, T.H., Yang, J., Flanagan, C., Solar-Lezama, A.: Faceted execution of policy-agnostic programs. In: PLAS (2013)
8. Balliu, M., Dam, M., Guernic, G.L.: ENCoVer: Symbolic Exploration for Information Flow Security. In: CSF (2012)
9. Balliu, M., Dam, M., Le Guernic, G.: Epistemic Temporal Logic for Information Flow Security. In: PLAS (2011)
10. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive Declassification Policies and Modular Static Enforcement. In: S & P (2008)
11. Bao, T., Zheng, Y., Lin, Z., Zhang, X., Xu, D.: Strict control dependence and its effect on dynamic information flow analyses. In: ISSTA (2010)
12. Barthe, G., Crespo, J.M., Devriese, D., Piessens, F., Rivas, E.: Secure multi-execution through static program transformation. In: IFIP (2012)
13. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. MSCS (2011)
14. We are family: Relating information flow trackers (Extended Version). `http://www.cse.chalmers.se/research/group/security/family`
15. Beringer, L.: End-to-end multilevel hybrid information flow control. In: Jhala, R., Igarashi, A. (eds.) APLAS (2012)
16. Biba, K.J.: Integrity considerations for secure computer systems. Tech. rep., MITRE Corp. (1977)
17. Bielova, N., Rezk, T.: A taxonomy of information flow monitors. In: POST (2016)
18. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: CSF (2010)
19. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: PLDI (2009)
20. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: SPC (2005)
21. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM (1977)
22. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: S&P 2010 (2010)
23. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: VMCAI (2012)
24. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (2014)
25. Ernst, M.D., Just, R., Millstein, S., Dietl, W., Pernsteiner, S., Roesner, F., Koscher, K., Barros, P.B., Bhoraskar, R., Han, S., Vines, P., Wu, E.X.: Collaborative verification of information flow for a high-assurance app store. In: CCS (2014)
26. Fenton, J.S.: Memoryless subsystems. Comput. J. 17(2), 143–147 (1974)
27. Giacobazzi, R., Mastroeni, I.: Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. In: POPL (2004)
28. Goguen, J.A., Meseguer, J.: Security policies and security models. In: S&P (1982)
29. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS (2015)
30. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: SAC (2014)

31. Hedin, D., Bello, L., Sabelfeld, A.: Value-sensitive hybrid information flow control for a javascript-like language. In: CSF (2015)
32. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL. pp. 79–90 (2006)
33. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in JavaScript web applications. In: CCS (2010)
34. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: dynamic taint analysis with targeted control-flow propagation. In: NDSS (2011)
35. King, D., Hicks, B., Hicks, M., Jaeger, T.: Implicit flows: Can't live with 'em, can't live without 'em. In: ICISS (2008)
36. Le Guernic, G.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis, Kansas State University (2007)
37. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly inlining of dynamic security monitors. In: Computers & Security (2010)
38. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: S&P (1994)
39. Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: CSF (2011)
40. Nanevski, A., Banerjee, A., Garg, D.: Dependent type theory for verification of information flow and access control policies. ACM Trans. Program. Lang. (2013)
41. https://f-droid.org/repository/browse/?fdid=name.bagi.levente.pedometer
42. Russo, A., Sabelfeld, A., Li, K.: Implicit flows in malicious and nonmalicious code. Marktoberdorf Summer School (IOS Press) (2009)
43. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: CSF (2010)
44. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. JSAC (2003)
45. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. JCS (2009)
46. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: ISSS (2003)
47. Schoepe, D., Balliu, M., Pierce, B.C., Sabelfeld, A.: Explicit secrecy: A policy for taint tracking. In: EuroS&P (2016)
48. Schoepe, D., Balliu, M., Piessens, F., Sabelfeld, A.: Let's face it: Faceted values for taint tracking. In: ESORICS (2016)
49. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: S&P 2010 (2010)
50. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: CSF (2007)
51. SnoopWall: Flashlight Apps Threat Assessment Report. https://www.snoopwall.com/reports (2014)
52. Staicu, C., Pradel, M.: An empirical study of implicit information flow (2015), poster at PLDI. https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_SOLA/Papers/poster-pldi2015-src.pdf
53. http://www.heartbleed.com (2015)
54. Tripp, O., Ferrara, P., Pistoia, M.: Hybrid security analysis of web javascript code via dynamic partial evaluation. In: ISSTA (2014)
55. 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. In: MICRO (2004)
56. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: CASCR (1999)
57. Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: ICICS (2006)

58. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. JCS (1996)
59. Volpano, D.M.: Safety versus secrecy. In: SAS (1999)
60. Zdancewic, S.A.: Programming Languages for Information Security. Ph.D. thesis, Cornell University, Ithaca, NY, USA (2002)

A Semantics

In this section we present the small-step operational semantics of SIMPL, and the instrumented semantics used to extract programs for weak secrecy as introduced in Section 2.4.

A.1 Operational Semantics of SIMPL

We present the semantics of the language in terms of memories and environments. We evaluate expressions in the context of a memory as expected.

We comment on a few interesting rules in Figure 10. Rule E-Assign evaluates an expression e in the context of a memory m and yields a new memory m' where only variable x is assigned the value n, otherwise m and m' are the same. Rule E-In reads the top value from the environment stream $\mathcal{E}$ associated with security level ℓ and updates the memory by assigning the value to variable x. Moreover, it updates the stream by removing the value just read. Rule E-Out evaluates an expression e in the context of a memory m and outputs the resulting value on channel ℓ. Rules E-IfTrue and E-IfFalse evaluate the guard e in the context of current memory m and execute the corresponding statements.

A.2 Instrumented Semantics for Weak and Observable secrecy

In this subsection we provide the full set of rules about how the extraction context makes use of the operational semantics to extract programs for weak secrecy. Fig. 11 presents an instrumented semantics that transforms contexts for weak secrecy.

B Proofs for Enforcement Mechanisms

In this section we provide soundness proofs for the enforcement mechanisms presented in Section 3.

We first show the instrumented semantics is *faithful* to the standard semantics in Fig. 10 in the sense that whenever an instrumented execution reaches a program state, it is possible for the standard execution to reach the same state. We sometimes write $\overset{\tau}{\underset{\mathcal{I}}{\twoheadrightarrow}}$ to denote an instrumented transition for dynamic tracker $\mathcal{I} \in \{wt, ot, ft\}$.

We define successful executions for the standard semantics (Fig. 10), and for the instrumented semantics with a given instantiation $\mathcal{I}$ (Fig. 7 and Table 1). Successful executions (instrumented or not) never lead to abnormal program termination.

Definition 7 (Successful instrumented execution). *For any instantiation $\mathcal{I}$, program P, initial state $(\mathcal{E}, m)$, initial security environment Γ, and program counter stack pc, an execution is successful, written $\mathcal{I} \vdash_{\mathcal{E},m} P$, if $pc, \Gamma, \mathcal{E} \vdash \langle P, m\rangle \not\twoheadrightarrow^* \lightning$.*

E-Skip

$$\frac{}{\mathcal{E} \vdash \langle \mathbf{skip}, m \rangle \to \mathcal{E} \vdash \langle \varepsilon, m \rangle}$$

E-Assign

$$\frac{m(e) = n}{\mathcal{E} \vdash \langle x := e, m \rangle \to \mathcal{E} \vdash \langle \varepsilon, m[x \mapsto n] \rangle}$$

E-Out

$$\frac{m(e) = n}{\mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m \rangle \xrightarrow{[(\ell,n)]} \mathcal{E} \vdash \langle \varepsilon, m \rangle}$$

E-Seq

$$\frac{\mathcal{E} \vdash \langle P_1, m \rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1', m' \rangle}{\mathcal{E} \vdash \langle P_1 \; ; P_2, m \rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1' \; ; P_2, m' \rangle}$$

E-SeqEmpty

$$\frac{}{\mathcal{E} \vdash \langle \varepsilon \; ; P_2, m \rangle \to \mathcal{E} \vdash \langle P_2, m \rangle}$$

E-IfTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m \rangle \to \mathcal{E} \vdash \langle P_1, m \rangle}$$

E-IfFalse

$$\frac{m(e) = \mathbf{ff}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m \rangle \to \mathcal{E} \vdash \langle P_2, m \rangle}$$

E-WhileFalse

$$\frac{m(e) = \mathbf{ff}}{\mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m \rangle \to \mathcal{E} \vdash \langle \varepsilon, m \rangle}$$

E-WhileTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m \rangle \to \mathcal{E} \vdash \langle P \; ; \mathbf{while}\ e\ \mathbf{do}\ P, m \rangle}$$

E-In

$$\frac{\mathcal{E}' = \mathcal{E}[\ell \mapsto n \mapsto \mathcal{E}(\ell)(n+1)] \qquad m' = m[x \mapsto \mathcal{E}(\ell)(0)]}{\mathcal{E} \vdash \langle x \leftarrow \mathbf{in}(\ell), m \rangle \to \mathcal{E}' \vdash \langle \varepsilon, m' \rangle}$$

Fig. 10. Operational Semantics of SIMPL

We can easily show that the small-step operational semantics in Fig. 10, and the extended semantics in Fig. 11 and Fig. 6 produce the same executions, modulo the context C.

Lemma 1 (Equivalence). *Given a program P, an initial state $(\mathcal{E}, m)$ then $\mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m' \rangle$ iff $\mathcal{E} \vdash \langle P, m, [] \rangle \xrightarrow[\mathcal{I}]{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m', C \rangle$ for some context C and $\mathcal{I} \in \{wt, ot\}$.*

Proof. By rule inspection and the fact that contexts never cause abnormal termination.

B.1 Proofs for Weak Tracking

We first show that the instrumented semantics is *faithful* to the original semantics.

Lemma 2 (Faithfulness). *Given a program P, an initial state $(\mathcal{E}, m)$ and a security environment Γ, if $\Gamma, \bot, \mathcal{E} \vdash \langle P, m \rangle \overset{\tau}{\underset{wt}{\twoheadrightarrow}}{}^{*} \Gamma', \bot, \mathcal{E}' \vdash \langle P', m' \rangle$, then $\mathcal{E} \vdash \langle P, m, [] \rangle \xrightarrow{\tau}{}^{*} \mathcal{E}' \vdash \langle P', m', C \rangle$.*

W-Skip

$$\frac{}{\mathcal{E} \vdash \langle \mathbf{skip}, m, C\rangle \rightarrow \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{skip}]\rangle}$$

W-SeqEmpty

$$\frac{}{\mathcal{E} \vdash \langle \varepsilon \; ; P_2, m, C\rangle \rightarrow \mathcal{E} \vdash \langle P_2, m, C \; ; []\rangle}$$

W-Assign

$$\frac{m(e) = n}{\mathcal{E} \vdash \langle x := e, m, C\rangle \rightarrow \mathcal{E} \vdash \langle \varepsilon, m[x \mapsto n], C[x := e]\rangle}$$

W-Out

$$\frac{m(e) = n}{\mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m, C\rangle \xrightarrow{[(\ell,n)]} \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{out}(\ell, e)]\rangle}$$

W-In

$$\frac{\mathcal{E}' = \mathcal{E}[\ell \mapsto n \mapsto \mathcal{E}(\ell)(n+1)] \qquad m' = m[x \mapsto \mathcal{E}(\ell)(0)]}{\mathcal{E} \vdash \langle x \leftarrow \mathbf{in}(\ell), m, C\rangle \rightarrow \mathcal{E}' \vdash \langle \varepsilon, m', C[x \leftarrow \mathbf{in}(\ell)]\rangle}$$

W-Seq

$$\frac{\mathcal{E} \vdash \langle P_1, m, C\rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1', m', C'\rangle}{\mathcal{E} \vdash \langle P_1 \; ; P_2, m, C\rangle \xrightarrow{\alpha} \mathcal{E}' \vdash \langle P_1' \; ; P_2, m', C'\rangle}$$

W-IfTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m, C\rangle \rightarrow \mathcal{E} \vdash \langle P_1, m, C\rangle}$$

W-IfFalse

$$\frac{m(e) = \mathbf{ff}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, m, C\rangle \rightarrow \mathcal{E} \vdash \langle P_2, m, C\rangle}$$

W-WhileFalse

$$\frac{m(e) = \mathbf{ff}}{\mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m, C\rangle \rightarrow \mathcal{E} \vdash \langle \varepsilon, m, C[\mathbf{skip}]\rangle}$$

W-WhileTrue

$$\frac{m(e) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ P, m, C\rangle \rightarrow \mathcal{E} \vdash \langle P \; ; \mathbf{while}\ e\ \mathbf{do}\ P, m, C\rangle}$$

Fig. 11. Weak Secrecy

Proof. By rule inspection. The rules for weak tracking (Fig. 7) are a constraint version of the operational semantics rules (Fig. 10 and Fig. 11).

To simplify notation, we write $\Gamma, \bot, \mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow{\tau}\!\!\twoheadrightarrow^* \Gamma', \bot, \mathcal{E}' \vdash \langle P', m', C'\rangle$ to denote both the instrumented execution $\Gamma, \bot, \mathcal{E} \vdash \langle P, m\rangle \xrightarrow{\tau}\!\!\twoheadrightarrow^* \Gamma', \bot, \mathcal{E}' \vdash \langle P', m'\rangle$ and the corresponding concrete execution $\mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m', C\rangle$. We also assume a two point lattice where $\mathbf{L} \sqsubseteq \mathbf{H}$.

Lemma 3. *Let $\Gamma, \bot, \mathcal{E} \vdash \langle P, m, [] \rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma', \bot, \mathcal{E}' \vdash \langle P', m', C \rangle$ and $P^* = C[\textbf{\textit{skip}}]$ then for all m^*, $m^* \approx_{\Gamma} m$ and $\mathcal{E}^* \approx_{\ell} \mathcal{E}$, we have $\mathcal{E}^* \vdash \langle P^*, m^* \rangle \xrightarrow{\tau'^*}^* \mathcal{E}'^* \vdash \langle \varepsilon, m'^* \rangle$ and $\mathcal{E} \vdash \langle P^*, m \rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle \varepsilon, m' \rangle$ and $\mathcal{E}'^* \approx_{\ell} \mathcal{E}'$, $m'^* \approx_{\Gamma'} m'$ and $\tau'^* \approx_{\ell} \tau$.*

Proof. We proceed by induction on $\overset{\tau}{\twoheadrightarrow}^*$ and P^*:

Reflexive case: We have $P^* = \varepsilon$, $\mathcal{E} = \mathcal{E}'$, $m = m'$ and $\tau = \tau'$, hence the statements holds trivially.

Transitive case: Assume $\Gamma, \bot, \mathcal{E} \vdash \langle P, m, [] \rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma'', \bot, \mathcal{E}'' \vdash \langle P'', m'', C'' \rangle \overset{\alpha}{\twoheadrightarrow} \Gamma', \bot, \mathcal{E}' \vdash \langle P', m', C \rangle$ and $P^* = \mathcal{T}_{\mathcal{WS}}(P, P'', \mathcal{E}, m)$; $\mathcal{T}_{\mathcal{WS}}(P'', P', \mathcal{E}, m'')$. Let $P^* = C[\textbf{skip}]$ and $P_1 = C''[\textbf{skip}]$.

By induction hypothesis it holds that for all $\mathcal{E}^*, m^*$, $m^* \approx_{\Gamma} m$ and $\mathcal{E}^* \approx_{\ell} \mathcal{E}$, we have $\mathcal{E}^* \vdash \langle P_1, m^* \rangle \xrightarrow{\tau''^*}^* \mathcal{E}''^* \vdash \langle \varepsilon, m''^* \rangle$ and $\mathcal{E} \vdash \langle P_1, m \rangle \xrightarrow{\tau''}^* \mathcal{E}'' \vdash \langle \varepsilon, m'' \rangle$ and $\mathcal{E}''^* \approx_{\ell} \mathcal{E}''$, $m''^* \approx_{\ell} m''$ and $\tau''^* \approx_{\ell} \tau''$. We show the inductive step by case analysis on P'' and rules in Fig. 7 and Table 1, and rules in Fig. 11. The only interesting cases are rules S-Assign, S-In and S-Out, since the other rules will never change the context.

Case S-Assign: If $P'' = x := e$ we have $P^* = P_1$; $x := e$, then $m' = m''[x \mapsto m''(e)]$ and $\Gamma' = \Gamma''[x \mapsto \Gamma''(e)]$, and $m'^* = m''^*[x \mapsto m''^*(e)]$. Moreover, $\mathcal{E}' = \mathcal{E}''$ and $\tau' = \tau''$. We show that $m' \approx_{\Gamma'} m'^*$. There are 2 cases to consider: (i) if $\Gamma'(e) = \mathbf{H}$ then $\Gamma'(x) = \mathbf{H}$, hence $m' \approx_{\Gamma'} m'^*$. (ii) otherwise $\Gamma'(e) = \mathbf{L}$, which implies that $m'(x) = m'^*(x)$, i.e., $m' \approx_{\Gamma'} m'^*$.

Case S-In: If $P'' = x \leftarrow \mathbf{in}(\ell)$ we have $P^* = P_1$; $x \leftarrow \mathbf{in}(\ell)$, then $m' = m''[x \mapsto \mathcal{E}''(\ell)(0)]$, $\mathcal{E}' = \mathcal{E}''[\ell \mapsto n \mapsto \mathcal{E}''(\ell)(n+1)]$ and $\Gamma' = \Gamma''[x \mapsto \ell]$, and $m'^* = m''^*[x \mapsto \mathcal{E}''^*(\ell)(0)]$ and $\mathcal{E}'^* = \mathcal{E}''^*[\ell \mapsto n \mapsto \mathcal{E}''^*(\ell)(n+1)]$. Clearly $\tau' = \tau''$. We show that (i) $m' \approx_{\Gamma'} m'^*$ and (ii) $\mathcal{E}' \approx_{\ell} \mathcal{E}'^*$. Case (i): if $\ell = \mathbf{H}$ then $\Gamma'(x) = \mathbf{H}$, hence $m' \approx_{\Gamma'} m'^*$. Otherwise $\ell = \mathbf{L}$, then $\mathcal{E}''(\ell)(0) = \mathcal{E}''^*(\ell)(0)$ by hypothesis, hence $m'(x) = m'^*(x)$, i.e., $m' \approx_{\Gamma'} m'^*$. Case (ii) if $\ell = \mathbf{H}$ then, by the hypothesis $\mathcal{E}'' \approx_{\ell} \mathcal{E}''^*$, therefore $\mathcal{E}' \approx_{\ell} \mathcal{E}'^*$.

Case S-Out: If $P'' = \mathbf{out}(\ell, e)$ we have $P^* = P_1$; $\mathbf{out}(\ell, e)$, then $m' = m''$ and $\mathcal{E}' = \mathcal{E}''$. We show that $\tau' \approx_{\ell} \tau'^*$. If $\ell = \mathbf{H}$, we are done. Otherwise $m''(e) = m''^*(e)$, since $\Gamma''(e) \sqsubseteq \mathbf{L}$, hence $\tau' \approx_{\ell} \tau'^*$.

Proof (Proof for Thrm. 1). We show that given a state $(\mathcal{E}, m)$ and a security environment Γ, if $\Gamma, \bot, \mathcal{E} \vdash \langle P, m \rangle \xrightarrow{\tau}^* \Gamma', \bot, \mathcal{E}' \vdash \langle P', m' \rangle$ then $\mathcal{E} \vdash \langle P, m, [] \rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m', C \rangle$ and $\mathit{WS} \vDash_{\mathcal{E}, m} C[\textbf{skip}]$, i.e. for all $(\mathcal{E}_0, m_0)$, $\mathcal{E}_0 \vdash \langle C[\textbf{skip}], m_0 \rangle \xrightarrow{\tau.\alpha}^* \mathcal{E}^* \vdash \langle P^*, m^* \rangle$, then we have that $k_P(C[\textbf{skip}], \mathcal{E}_0, m_0, \tau) = k(C[\textbf{skip}], \mathcal{E}_0, m_0, \tau.\alpha)$. The first part follows from Lemma 2. The second part follows from Lemma 3.

B.2 Proofs for Observable Tracking

We prove that any program that terminates normally under the instrumented semantics for observable tracking, i.e. $\mathcal{I} = \mathit{OS}$, satisfies observable secrecy.

Lemma 4 (Faithfulness). *Given a program P, an initial state $(\mathcal{E}, m)$ and a security environment Γ, if $\Gamma, \bot, \mathcal{E} \vdash \langle P, m \rangle \underset{ot}{\overset{\tau}{\twoheadrightarrow}}^* \Gamma', pc', \mathcal{E}' \vdash \langle P', m' \rangle$, then $\mathcal{E} \vdash \langle P, m, [] \rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m', C \rangle$.*

Proof. By rule inspection. The rules for weak tracking (Fig. 7) are a constraint version of the operational semantics rules (Fig. 10 and Fig. 6).

To simplify notation, we write $\Gamma, \bot, \mathcal{E} \vdash \langle P, m, []\rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma', pc', \mathcal{E}' \vdash \langle P', m', C'\rangle$ to denote both the instrumented execution $\Gamma, \bot, \mathcal{E} \vdash \langle P, m\rangle \underset{ot}{\overset{\tau}{\twoheadrightarrow}}^* \Gamma', pc', \mathcal{E}' \vdash \langle P', m'\rangle$ and the corresponding concrete execution $\mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m', C\rangle$. We also assume a two point lattice where $\mathbf{L} \sqsubseteq \mathbf{H}$.

Lemma 5. *Let $\Gamma, \bot, \mathcal{E} \vdash \langle P, m, []\rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma', pc', \mathcal{E}' \vdash \langle P', m', C\rangle$ and $P^* = C[\textbf{\textit{skip}}]$ then for all m^*, $m^* \approx_\Gamma m$ and $\mathcal{E}^* \approx_\ell \mathcal{E}$, we have $\mathcal{E}^* \vdash \langle P^*, m^*\rangle \xrightarrow{\tau'^*}^* \mathcal{E}'^* \vdash \langle \varepsilon, m'^*\rangle$ and $\mathcal{E} \vdash \langle P^*, m\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle \varepsilon, m'\rangle$ and $\mathcal{E}'^* \approx_\ell \mathcal{E}'$, $m'^* \approx_{\Gamma'} m'$ and $\tau'^* \approx_\ell \tau$.*

Proof. We proceed by induction on $\overset{\tau}{\twoheadrightarrow}^*$ and P^*:

Reflexive case: We have $P^* = \varepsilon$, $\mathcal{E} = \mathcal{E}'$, $m = m'$ and $\tau = \tau'$, hence the statements holds trivially.

Transitive case: Assume $\Gamma, \bot, \mathcal{E} \vdash \langle P, m, []\rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma'', pc'', \mathcal{E}'' \vdash \langle P'', m'', C''\rangle \overset{\alpha}{\twoheadrightarrow} \Gamma', pc', \mathcal{E}' \vdash \langle P', m', C\rangle$ and $P^* = C[\mathbf{skip}]$. Let also $P_1 = C''[\mathbf{skip}]$.

By induction hypothesis it holds that for all $\mathcal{E}^*, m^*$, $m^* \approx_\Gamma m$ and $\mathcal{E}^* \approx_\ell \mathcal{E}$, we have $\mathcal{E}^* \vdash \langle P_1, m^*\rangle \xrightarrow{\tau''^*}^* \mathcal{E}''^* \vdash \langle \varepsilon, m''^*\rangle$ and $\mathcal{E} \vdash \langle P_1, m\rangle \xrightarrow{\tau''}^* \mathcal{E}'' \vdash \langle \varepsilon, m''\rangle$ and $\mathcal{E}''^* \approx_\ell \mathcal{E}''$, $m''^* \approx_\ell m''$ and $\tau''^* \approx_\ell \tau''$. We show the inductive step by case analysis on P'' and rules in Fig. 7 and Table 1, and Fig. 6. We report a few interesting cases.

Case S-Assign: If $P'' = x := e$ we have $P^* = C[x := e]$, then $m' = m''[x \mapsto m''(e)]$ and $\Gamma' = \Gamma''[x \mapsto pc \sqcup \Gamma''(e)]$, and $m'^* = m''^*[x \mapsto m''^*(e)]$. Moreover, $\mathcal{E}' = \mathcal{E}''$ and $\tau' = \tau''$. We show that $m' \approx_{\Gamma'} m'^*$. There are 2 cases to consider: (i) if $\Gamma'(e) = \mathbf{H}$ or $pc = \mathbf{H}$ then $\Gamma'(x) = \mathbf{H}$, hence $m' \approx_{\Gamma'} m'^*$. (ii) otherwise $\Gamma'(e) = pc = \mathbf{L}$, which implies that $m'(x) = m'^*(x)$, i.e., $m' \approx_{\Gamma'} m'^*$.

Case S-In: If $P'' = x \leftarrow \mathbf{in}(\ell)$ we have $P^* = C[x \leftarrow \mathbf{in}(\ell)]$, then $m' = m''[x \mapsto \mathcal{E}''(\ell)(0)]$, $\mathcal{E}' = \mathcal{E}''[\ell \mapsto n \mapsto \mathcal{E}''(\ell)(n+1)]$, and $\Gamma' = \Gamma''[x \mapsto \ell]$ (since $pc \sqsubseteq \ell$), and $m'^* = m''^*[x \mapsto \mathcal{E}''^*(\ell)(0)]$, and $\mathcal{E}'^* = \mathcal{E}''^*[\ell \mapsto n \mapsto \mathcal{E}''^*(\ell)(n+1)]$. Clearly $\tau' = \tau''$. We show that (i) $m' \approx_{\Gamma'} m'^*$ and (ii) $\mathcal{E}' \approx_\ell \mathcal{E}'^*$. Case (i): if $\ell = \mathbf{H}$ then $\Gamma'(x) = \mathbf{H}$, hence $m' \approx_{\Gamma'} m'^*$. Otherwise $\ell = \mathbf{L}$, then $\mathcal{E}''(\ell)(0) = \mathcal{E}''^*(\ell)(0)$ by hypothesis, hence $m'(x) = m'^*(x)$, i.e., $m' \approx_{\Gamma'} m'^*$. Case (ii) If $\ell = \mathbf{L}$, by the hypothesis $\mathcal{E}'' \approx_\ell \mathcal{E}''^*$, therefore $\mathcal{E}' \approx_\ell \mathcal{E}'^*$.

Case S-Out: If $P'' = \mathbf{out}(\ell, e)$ we have $P^* = C[\mathbf{out}(\ell, e)]$, then $m' = m''$ and $\mathcal{E}' = \mathcal{E}''$. We show that $\tau' \approx_\ell \tau'^*$. If $\ell = \mathbf{H}$, we are done. Otherwise $m''(e) = m''^*(e)$, since $\Gamma''(e) \sqsubseteq \mathbf{L}$ and $pc = \mathbf{L}$, hence $\tau' \approx_\ell \tau'^*$.

Case S-IfTrue: If $P'' = \mathbf{if}\ e\ \mathbf{then}\ P_{21}\ \mathbf{else}\ P_{22}$, then, assuming that $m''(e) = \mathbf{tt}$, $C = C''[\mathbf{if}\ e\ \mathbf{then}\ []\ \mathbf{else}\ \mathbf{skip}]$ and $P^* = C''[\mathbf{if}\ e\ \mathbf{then}\ \mathbf{skip}\ \mathbf{else}\ \mathbf{skip}]$. Then the claim follows from the induction hypothesis, since $\mathbf{if}\ e\ \mathbf{then}\ \mathbf{skip}\ \mathbf{else}\ \mathbf{skip}$ is semantically equivalent to $\mathbf{skip}$ and $C''[\mathbf{skip}] = C[\mathbf{skip}]$. The proof for S-IfFalse is symmetric.

Proof (Proof for Thrm 2). We show that a state $(\mathcal{E}, m)$ and a security environment Γ, if $\Gamma, \bot, \mathcal{E} \vdash \langle P, m\rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma', pc', \mathcal{E}' \vdash \langle P', m'\rangle$ then $\mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash$

$\langle P', m', C\rangle$ and $OS \vDash_{\mathcal{E},m} C[\textbf{skip}]$, i.e. for all $(\mathcal{E}_0, m_0)$, $\mathcal{E}_0 \vdash \langle C[\textbf{skip}], m_0\rangle \xrightarrow{\tau.\alpha}^{*}$ $\mathcal{E}^* \vdash \langle P^*, m^*\rangle$, then we have that $k_P(C[\textbf{skip}], \mathcal{E}_0, m_0, \tau) = k(C[\textbf{skip}], \mathcal{E}_0, m_0, \tau.\alpha)$. The first part follows from Lemma 4. The second part follows from Lemma 5.

B.3 Proofs for Full Tracking

We show that any program that terminates normally under the instrumented semantics for full tracking, i.e. $\mathcal{I} = FS$, satisfies full secrecy.

Lemma 6 (Faithfulness). *Given a program P, an initial state $(\mathcal{E}, m)$ and a security environment Γ, if $\Gamma, \bot, \mathcal{E} \vdash \langle P, m\rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma', pc', \mathcal{E}' \vdash \langle P', m'\rangle$, then $\mathcal{E} \vdash \langle P, m, []\rangle \xrightarrow[ft]{\tau}^{*} \mathcal{E}' \vdash \langle P', m', C\rangle$.*

Proof (Proof for Thrm. 3). Similar to Theorem 7.

C Staged Information-Flow Control

We introduce a family of shadow fresh variables $pc, pc', pc'', \cdots$ to emulate the program counter stack from Table 1. In particular, we use variable pc to store the current security context, and *primed* variables to save and restore the security context after conditional and loop statements. We use pc^* to denote a fresh primed variable. We assume an interpretation of lattice operators $(\sqsubseteq, \sqcup, \sqcap)$ in the language semantics. The semantics of lattice operators and security levels can be embedded in the language using, for instance, bitwise operations as in TaintDroid [24]. We write $;\{\mathcal{P}'\}_\phi$ to denote the sequential composition of a set of programs $\mathcal{P}' \subseteq \mathcal{P}$ satisfying boolean condition ϕ.

For each security level ℓ in the original lattice $\mathcal{L}$, we introduce an additional level $\widetilde{\ell}$ used to distinguish the outputs performed by our transformation from the outputs of the original program. The output of the transformation uses the security lattice $\mathcal{L}' = \mathcal{L} \cup \{\widetilde{\ell} | \ell \in \mathcal{L}\}$ with $\widetilde{\ell} \sqsubseteq \ell'$ iff $\ell \sqsubseteq \ell'$. The new lattice replicates the security levels of original lattice, respecting the ordering between security levels. As we will see, our transformations introduce additional output commands on channels with security level $\widetilde{\ell}$ to enable a weak tracker capture observable and/or hidden implicit flows.

For a program P and an enforcement $\mathcal{I}$, we give a source-to-source transformation $\mathcal{T}_{\mathcal{I}} : \mathcal{P} \rightarrow \mathcal{P}$. We use $\lightning$ to cause the program terminate abnormally due to a security exception thrown by the weak tracker.

Leveraging Weak Tracking for Observable Secrecy We now present a program transformation to enable weak tracking enforce observable secrecy. The transformation is transparent to security labels which we can specify when the transformed program is executed on a weak tracker. The main advantage of this approach is that it only requires a simple inlining the program counter stack (cf. Table 1) into the code of the original program.

Since program counter variable pc is only used to track the context security level, we write $\iota(pc)$ for an expression over pc that is identity-preserving with respect to an operator $\oplus$. This causes a weak tracker to account for the security level of the pc without changing the semantics of the original expression. For instance, in the assignment $x := e + \iota(pc)$, we can define $\iota(pc)$ as $pc - pc$, thus preserving the semantics of the assignment. We will show in Sect. 5 how these operations are implemented for Android apps running on a real weak tracker (TaintDroid). We extend the initial security environment Γ of the weak tracker by mapping to $\perp$ all fresh variables introduced by the transformation.

$$
\begin{aligned}
&\mathcal{T}_{\mathcal{OT}}(\mathbf{skip}) = \mathbf{skip}\\
&\mathcal{T}_{\mathcal{OT}}(x := e) = x := e \oplus \iota(pc)\\
&\mathcal{T}_{\mathcal{OT}}(x \leftarrow \mathbf{in}(\ell)) = \mathbf{out}(\widetilde{\ell}, pc) \;;\; x \leftarrow \mathbf{in}(\ell) \;;\; x := x \oplus \iota(pc)\\
&\mathcal{T}_{\mathcal{OT}}(\mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) = pc^* := pc \;;\; pc := e \oplus \iota(pc) \;;\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ e\ \mathbf{then}\ \mathcal{T}_{\mathcal{OT}}(P_1)\ \mathbf{else}\ \mathcal{T}_{\mathcal{OT}}(P_2) \;;\; pc := pc^*\\
&\mathcal{T}_{\mathcal{OT}}(\mathbf{while}\ e\ \mathbf{do}\ c) = pc^* := pc \;;\; pc := e \oplus \iota(pc) \;;\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{while}\ e\ \mathbf{do}\ \mathcal{T}_{\mathcal{OT}}(c) \;;\; pc := pc^*\\
&\mathcal{T}_{\mathcal{OT}}(P_1 \;;\; P_2) = \mathcal{T}_{\mathcal{OT}}(P_1) \;;\; \mathcal{T}_{\mathcal{OT}}(P_2)\\
&\mathcal{T}_{\mathcal{OT}}(\mathbf{out}(\ell, e)) = \mathbf{out}(\ell, e \oplus \iota(pc))
\end{aligned}
$$

The transformation inlines the program counter variable for assignment and input statements to track observable implicit flows. We use security level replicas to check whether or not an input is allowed in a given security context, thus enabling a weak tracker to prevent observable implicit flows. In case of conditionals/loops, we save the program counter variable before entering a conditional/loop statement, and restore it back whenever the conditional is exit; we assume that pc^* is fresh for each conditional/loop statement. The transformation of an output statement inlines the variable pc to capture observable implicit flows .

Example 2. Let $P = \mathbf{if}\ h\ \mathbf{then}\ \mathbf{out}(\mathbf{L}, \mathbf{tt})\ \mathbf{else}\ \mathbf{out}(\mathbf{L}, \mathbf{ff})$ be a program and h a boolean variable with security label $\mathbf{H}$. P leaks the value of variable h through an observable implicit flow. The transformation $\mathcal{T}_{\mathcal{OT}}(P)$ yields the following program, where initially $\Gamma(pc) = \Gamma(pc') = \mathbf{L}$ and $\Gamma(h) = \mathbf{H}$.

$$
\begin{aligned}
&pc' := pc \;;\; pc := h \oplus \iota(pc) \;;\\
&\mathbf{if}\ h\ \mathbf{then}\ \mathbf{out}(\mathbf{L}, \mathbf{tt} \oplus \iota(pc))\ \mathbf{else}\ \mathbf{out}(\mathbf{L}, \mathbf{ff} \oplus \iota(pc)) \;;\\
&pc := pc'
\end{aligned}
$$

Suppose $\iota(pc) = pc \vee pc$ and $\oplus = \vee$. We use the weak tracker in Figure 7 to run the transformed program. After executing the second assignment, we have $\Gamma'(pc) = \Gamma(h \oplus \iota(pc)) = \mathbf{H}$, therefore the execution of the output statement terminates the program abnormally, thus preventing the observable implicit flow.

We show that whenever the transformed program executes on the weak tracker in Figure 7, the original program satisfies observable secrecy for the same run.

Theorem 5. $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{OT}}(P) \Rightarrow OS \vDash_{\mathcal{E},m} P$

Leveraging Weak Tracking for Full Secrecy We now give a transformation that enables weak tracking to enforce full secrecy. We do this by inlining the program counter stack of full tracking monitor into the program's code. The transformation implements the NSU strategy by disallowing changes of security labels of variables under a high security context (in addition to preventing outputs in a high context).

$$\mathcal{T}_{\mathcal{FT}}(P) = \mathcal{T}_{\mathcal{OT}}(P) \text{ if } P \neq x := e$$
$$\mathcal{T}_{\mathcal{FT}}(x := e) = \mathbf{out}(\widetilde{\Gamma(x)}, e \oplus \iota(pc)) \ ; \ x := e \oplus \iota(pc)$$

The transformation leverages the extended lattice $\mathcal{L}'$ to ensure that assignment statements never change the security label of a variable in a high context. This is needed to prevent hidden implicit flows. Otherwise, the transformation is the same as the one for observable tracking. We show that whenever the transformed program executes on the weak tracker in Figure 7, the original program satisfies full secrecy for the same run.

Theorem 6. $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{FT}}(P) \Rightarrow FS \vDash_{\mathcal{E},m} P$

Cross Copy and Weak Tracking for Full Secrecy An alternative way of enforcing full secrecy is by *cross copying* the variables modified in each branch of a conditional or loop statement, and then using an observable tracker to monitor the transformed program. This would enable an observable tracker to raise the security labels of variables that do not appear in the branch that is currently executed by the monitor, and thus account for hidden implicit flows. The main advantage of this approach is that it boosts permissiveness in a way similar to observable tracking on the original program. Concretely, cross copying enables an observable tracking monitor to capture hidden implicit flows, without terminating potentially secure programs prematurely.

We define a function $MV : \mathcal{P} \mapsto 2^{Vars}$ that returns the set of modified variables for a given program P.

$$MV(P) = \emptyset \text{ if } P \in \{\mathbf{skip}, \mathbf{out}(\ell, e)\}$$
$$MV(x := e) = \{x\} \text{ if } P \in \{x := e, x \leftarrow \mathbf{in}(\ell)\}$$
$$MV(\mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) = MV(P_1) \cup MV(P_2)$$
$$MV(\mathbf{while}\ e\ \mathbf{do}\ P) = MV(P)$$
$$MV(P_1 \ ; P_2) = MV(P_1) \cup MV(P_2)$$

We use this function to define a transformation that, for each branch, cross copies the variables that are potentially modified in the other branch, thus enabling an observable tracker to raise their security level independently of whether or not those variables were modified in the executed branch of the original program. We can then use the transformation for observable tracking to leverage a weak tracker for full secrecy. As a result, both explicit and implicit flows will be prevented, as proved in Theorem 7.

$$\begin{aligned}
&\mathcal{T}_{\mathcal{MV}}(P) = P \text{ if } P \in \{\mathbf{skip}, x := e, x \leftarrow \mathbf{in}(\ell), \mathbf{out}(\ell, e)\}\\
&\mathcal{T}_{\mathcal{MV}}(\mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) =\\
&\qquad\qquad \mathbf{if}\ e\ \mathbf{then}\ P_1\ ;\ Id(MV(P_2) \setminus MV(P_1))\\
&\qquad\qquad\qquad \mathbf{else}\ P_2\ ;\ Id(MV(P_1) \setminus MV(P_2))\\
&\mathcal{T}_{\mathcal{MV}}(\mathbf{while}\ e\ \mathbf{do}\ P) = \mathbf{while}\ e\ \mathbf{do}\ \mathcal{T}_{\mathcal{MV}}(P)\ ;\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ !e\ \mathbf{then}\ Id(MV(P))\ \mathbf{else\ skip}\\
&\mathcal{T}_{\mathcal{MV}}(P_1\ ;\ P_2) = \mathcal{T}_{\mathcal{MV}}(P_1)\ ;\ \mathcal{T}_{\mathcal{MV}}(P_2)
\end{aligned}$$

We write $Id(S)$ for $;\{x := x\}_{x \in S}$, and use it to define a semantics-preserving transformation necessary for the security analysis. For conditional statements, the transformation cross copies every variable that is modified in the alternative branch as expected. For loop statements, the transformation is recursively applied to the loop body to enable detection of potential hidden flows therein. Moreover, since e is always false after exiting the loop, the subsequent conditional always evaluates to true. This allows to account for hidden flows whenever the loop guard evaluates to false.

We use program P from Ex. 3 to illustrate the approach. The transformation $\mathcal{T}_{\mathcal{MV}}(P)$ yields the following program.

$$\begin{aligned}
&l_1 := \mathbf{tt}\ ;\ l_2 := \mathbf{tt}\ ;\\
&\mathbf{if}\ h\ \mathbf{then}\ l_1 := \mathbf{ff}\ \mathbf{else}\ l_1 := l_1\\
&\mathbf{if}\ l_1\ \mathbf{then}\ l_2 := \mathbf{ff}\ \mathbf{else}\ l_2 := l_2\\
&\mathbf{out}(\mathbf{L}, l_2)
\end{aligned}$$

If we run the resulting program on the observable tracker in Figure 7, or, alternatively, transform and run $\mathcal{T}_{\mathcal{OT}}(\mathcal{T}_{\mathcal{MV}}(P))$ on a weak tracker, the execution will be terminated abnormally when the output statement is reached, thus preventing the hidden implicit flow present in the initial program. On the other hand, if the output statements were $\mathbf{out}(\mathbf{L}, 1)$ or $\mathbf{out}(\mathbf{H}, l_2)$, and therefore the programs were secure for full secrecy, they would be correctly accepted by the monitor. On the contrary, the full tracker in Thrm. 6 would incorrectly terminate both secure programs abnormally.

Theorem 7. $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{OT}}(\mathcal{T}_{\mathcal{MV}}(P)) \Rightarrow FS \vDash_{\mathcal{E},m} P$

D Proofs for Staged Analysis

D.1 Leveraging Weak Tracking for Observable Secrecy

Given a memory m and a transformation T, we write m^T for the memory m extended with the fresh variables introduced by transformation T. Moreover, given $T(P)$, we write m^T_P for the memory m^T restricted to the variables in P.

Lemma 7 (Faithfulness). *Let $m = m^T_P$, then whenever $\Gamma, \bot, \mathcal{E} \vdash \langle \mathcal{T}_{\mathcal{OT}}(P), m^T\rangle \overset{\tau}{\twoheadrightarrow}^* \Gamma', pc', \mathcal{E}' \vdash \langle \mathcal{T}_{\mathcal{OT}}(P)', m'^T\rangle$, then $\mathcal{E} \vdash \langle P, m\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m'\rangle$ and $m'^T_P = m'$.*

Proof. Induction on program P and $\mathcal{T}_{\mathcal{OT}}(P)$, similar to Lemma 8.

Lemma 8. *For any initial state $(\mathcal{E}, m)$, $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{OT}}(P) \Rightarrow OS \vdash_{\mathcal{E},m} P$.*

Proof. By induction on the structure of P. Let pc_w denote the fresh program variable used in $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{OT}}(P)$ and pc_o the program counter level used in $OS \vdash_{\mathcal{E},m} P$. Given a state $(\mathcal{E}, m)$, two security environments Γ_w and Γ_o with $\Gamma_w(pc_w) = pc_o$ and $\Gamma_w(x) = \Gamma_o(x)$ for all $x \in Var(P)$, we show that if $\Gamma_w, \mathcal{E} \vdash \langle \mathcal{T}_{\mathcal{OT}}(P), m\rangle \underset{wt}{\twoheadrightarrow}^* \Gamma'_w, \mathcal{E}' \vdash \langle \varepsilon, m'\rangle$ then $\Gamma_o, pc_o, \mathcal{E} \vdash \langle P, m_o\rangle \underset{ot}{\twoheadrightarrow}^* \Gamma'_o, pc'_o, \mathcal{E}' \vdash \langle \varepsilon, m'\rangle$ and $\Gamma'_w(pc_w) = pc'_o$ and $\Gamma'_w(x) = \Gamma'_o(x)$ for all $x \in Var(P)$. We instantiate $\iota(pc_w) = pc_w - pc_w$ and $\oplus = +$.

Case S-Skip: $P = \mathbf{skip}$, then $\mathcal{T}_{\mathcal{OT}}(P) = \mathbf{skip}$. It holds trivially.

Case S-Assign: $P = x := e$, then $\mathcal{T}_{\mathcal{OT}}(P) = x := e + pc_w - pc_w$. We have that $\Gamma'_w = \Gamma_w[x \mapsto \Gamma_w(e + pc_w - pc_w)]$ and $\Gamma'_o = \Gamma_o[x \mapsto \Gamma_o(e) \sqcup pc_o]$. By assumption $\Gamma_w(pc_w) = pc_o$ and $\Gamma_w(x) = \Gamma_o(x)$, hence $\Gamma_w(e + pc_w - pc_w) = \Gamma_o(e) \sqcup pc_o$, therefore $\Gamma'_o(x) = \Gamma'_w(x)$. The other invariants are trivially preserved.

Case S-If: $P = \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$, then $\mathcal{T}_{\mathcal{OT}}(P) = tpc_w := pc_w\ ;\ pc_w := e \oplus \iota(pc_w)\ ;\ \mathbf{if}\ e\ \mathbf{then}\ \mathcal{T}_{\mathcal{OT}}(P_1)\ \mathbf{else}\ \mathcal{T}_{\mathcal{OT}}(P_2)\ ;\ pc_w := tpc_w$. Suppose e evaluates to $\mathbf{tt}$. Then, before executing $\mathcal{T}_{\mathcal{OT}}(P_1)$, we have $\Gamma'_w(tpc_w) = \Gamma_w(pc_w)$ and $\Gamma'_w(pc_w) = \Gamma_w(e + pc_w - pc_w)$. Moreover, by the instrumented semantics of OS, we have $pc'_o = pc_o \sqcup \Gamma_o(e)$. This implies that $pc'_o = \Gamma'_w(pc'_w)$, since $pc_o \sqcup \Gamma_o(e) = \Gamma_w(pc_w - pc_w + e)$. By induction hypothesis, the execution of $\mathcal{T}_{\mathcal{OT}}(P_1)$ and P_1, implies $\Gamma''_w(pc_w) = pc''_o$. Finally, $\Gamma'''_w(pc) = pc'''_0$, by the initial assumption that $\Gamma_w(pc_w) = pc_o$. The other case where e evaluates to $\mathbf{ff}$ is symmetric.

Case S-Out: $P = \mathbf{out}(\ell, e)$, then $\mathcal{T}_{\mathcal{OT}}(P) = \mathbf{out}(\ell, e + pc - pc)$. The invariants are trivially preserved. Moreover, if $\underset{wt}{\twoheadrightarrow}$ succeeds, then $\Gamma_w(e + pc - pc) \sqsubseteq \ell$. This implies that $\Gamma_o(e) \sqsubseteq pc \sqcup \ell$, hence $\underset{ot}{\twoheadrightarrow}$ also will succeed.

Proof (Proof for Thrm. 5). Follows from Lemma 8 and Theorem 2.

D.2 Leveraging Weak Tracking for Full Secrecy

We prove that whenever the weak tracker in Sect. 3 terminates successfully on a given run of the transformed program, the original program satisfies full secrecy for the same run.

Lemma 9 (Faithfulness). *Let $m = m_P^T$, then whenever $\Gamma, \bot, \mathcal{E} \vdash \langle \mathcal{T}_{\mathcal{FT}}(P), m^T\rangle \xrightarrow{\tau}\!\!\!\!\rightarrow^* \Gamma', pc', \mathcal{E}' \vdash \langle \mathcal{T}_{\mathcal{FT}}(P)', m'^T\rangle$, then $\mathcal{E} \vdash \langle P, m\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m'\rangle$ and $m'^T_P = m'$.*

Proof. Induction on program P and $\mathcal{T}_{\mathcal{FT}}(P)$, similar to Lemma 10.

Lemma 10. *For any initial state $(\mathcal{E}, m)$, $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{FT}}(P) \Rightarrow FS \vdash_{\mathcal{E},m} P$.*

Proof. By induction on the structure of P. Let pc_w denote the fresh program variable used in $WS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{FS}}(P)$ and pc_f the program counter label used in $FS \vdash_{\mathcal{E},m} P$. Given a state $(\mathcal{E}, m)$, two security environments Γ_w and Γ_f with $\Gamma_w(pc_w) = pc_f$ and $\Gamma_w(x) = \Gamma_f(x)$ for all $x \in Var(P)$, we show that if $\mathit{\Gamma}_w, \mathcal{E} \vdash \langle \mathcal{T}_{\mathcal{FT}}(P), m\rangle \underset{wt}{\twoheadrightarrow}^* \mathit{\Gamma}'_w, \mathcal{E}' \vdash \langle \varepsilon, m'\rangle$ then $\mathit{\Gamma}_f, pc_f, \mathcal{E} \vdash \langle P, m_f\rangle \underset{ft}{\twoheadrightarrow}^* \mathit{\Gamma}'_f, pc'_f, \mathcal{E}' \vdash \langle \varepsilon, m'\rangle$ and $\mathit{\Gamma}'_w(pc_w) = pc'_f$ and $\mathit{\Gamma}'_w(x) = \mathit{\Gamma}'_f(x)$ for all $x \in Var(P)$. We instantiate $\iota(pc_w) = pc_w - pc_w$ and $\oplus = +$.

Case S-Assign: $P = x := e$, then $\mathcal{T}_{\mathcal{FT}}(P) = \mathbf{out}(\widetilde{\mathit{\Gamma}(x)}, e \oplus \iota(pc))$; $x := e \oplus \iota(pc)$. We have that $\mathit{\Gamma}'_w = \mathit{\Gamma}_w[x \mapsto \mathit{\Gamma}_w(e + pc_w - pc_w)]$ and $\mathit{\Gamma}'_f = \mathit{\Gamma}_f[x \mapsto \mathit{\Gamma}_f(e) \sqcup pc_f]$, and $\mathit{\Gamma}_w(pc_w) \sqcup \mathit{\Gamma}_w(e) \sqsubseteq \mathit{\Gamma}_w(x)$. By assumption $\mathit{\Gamma}_w(pc_w) = pc_f$ and $\mathit{\Gamma}_w(x) = \mathit{\Gamma}_f(x)$, hence $\mathit{\Gamma}_w(e+pc_w-pc_w) = \mathit{\Gamma}_f(e) \sqcup pc_f$, therefore $\mathit{\Gamma}'_f(x) = \mathit{\Gamma}'_w(x)$. Moreover, since $\mathit{\Gamma}_w(pc_w) \sqcup \mathit{\Gamma}_w(e) \sqsubseteq \mathit{\Gamma}_w(x)$, then $pc_f \sqcup \mathit{\Gamma}_f(e) \sqsubseteq \mathit{\Gamma}_f(x)$, hence $pc_f \sqsubseteq \mathit{\Gamma}_f(x)$.

Case S-Inp: $P = x \leftarrow \mathbf{in}(\ell)$, then $\mathcal{T}_{\mathcal{FT}}(P) = \mathbf{out}(\widetilde{\ell}, pc)$; $x \leftarrow \mathbf{in}(\ell)$; $x := x \oplus \iota(pc)$. We have that $\mathit{\Gamma}'_w = \mathit{\Gamma}_w[x \mapsto \ell \sqcup \mathit{\Gamma}_w(pc_w - pc_w)]$ and $\mathit{\Gamma}'_f = \mathit{\Gamma}_f[x \mapsto \ell \sqcup pc_f]$, and $\mathit{\Gamma}_w(pc_w) \sqsubseteq \ell$. By assumption $\mathit{\Gamma}_w(pc_w) = pc_f$ and $\mathit{\Gamma}_w(x) = \mathit{\Gamma}_f(x)$, hence $\ell \sqcup \mathit{\Gamma}_w(pc_w - pc_w) = \ell \sqcup pc_f$, therefore $\mathit{\Gamma}'_f(x) = \mathit{\Gamma}'_w(x)$. Moreover, since $\mathit{\Gamma}_w(pc_w) \sqsubseteq \ell$, then $pc_f \sqsubseteq \ell$.

Case S-If: $P = \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$, then $\mathcal{T}_{\mathcal{FT}}(P) = pc' := pc$; $pc := e \oplus \iota(pc)$; $\mathbf{if}\ e\ \mathbf{then}\ \mathcal{T}_{\mathcal{FT}}(P_1)\ \mathbf{else}\ \mathcal{T}_{\mathcal{FT}}(P_2)$; $pc := pc'$. Suppose e evaluates to $\mathbf{tt}$. Then, before executing $T(P_1)$, we have $\mathit{\Gamma}'_w(tpc_w) = \mathit{\Gamma}_w(pc_w)$ and $\mathit{\Gamma}'_w(pc_w) = \mathit{\Gamma}_w(pc_w - pc_w + e)$. By the instrumented semantics of FS, we have $pc'_f = pc_f \sqcup \mathit{\Gamma}_f(e)$. This implies that $pc'_f = \mathit{\Gamma}'_w(pc'_w)$, since $pc_f \sqcup \mathit{\Gamma}_f(e) = \mathit{\Gamma}_w(pc_w - pc_w + e)$. By induction hypothesis, the execution of $T(P_1)$ and P_1, implies $\mathit{\Gamma}''_w(pc_w) = pc''_f$. Finally, $\mathit{\Gamma}'''_w(pc) = pc'''_f$, by the initial assumption that $\mathit{\Gamma}_w(pc_w) = pc_f$. The other case where e evaluates to $\mathbf{ff}$ is symmetric.

Case S-Out: $P = \mathbf{out}(\ell, e)$, then $\mathcal{T}(P) = \mathbf{out}(\ell, e + \iota(pc))$. The invariants are trivially preserved. Moreover, if $\xrightarrow{wt}\!\!\!\!\rightarrow$ succeeds, then $\mathit{\Gamma}_w(e + pc_w - pc_w) \sqsubseteq \ell$. This implies that $\mathit{\Gamma}_f(e) \sqsubseteq pc_f \sqcup \ell$, hence $\xrightarrow{ft}\!\!\!\!\rightarrow$ also succeeds.

Proof (Proof for Thrm. 6). Follows from Lemma 10 and Theorem 3.

D.3 Static Analysis and Weak Tracking for Full Secrecy

We prove that whenever the weak tracker in Sect. 3 terminates successfully on a given run of the transformed program, the original program satisfies full secrecy for the same run of the original program.

Lemma 11 (Faithfulness). *Given a program P, an initial state $(\mathcal{E}, m)$ and a security environment $\mathit{\Gamma}$, if $\mathit{\Gamma}, pc, \mathcal{E} \vdash \langle \mathcal{T}_{\mathcal{MV}}(P), m\rangle \underset{ot}{\overset{\tau}{\twoheadrightarrow}}^* \mathit{\Gamma}', pc', \mathcal{E}' \vdash \langle \mathcal{T}_{\mathcal{MV}}(P)', m'\rangle$, then $\mathcal{E} \vdash \langle P, m\rangle \xrightarrow{\tau}^* \mathcal{E}' \vdash \langle P', m'\rangle$.*

Proof. By rule inspection. The rules for observable tracking (Fig. 7) are a constraint version of the operational semantics rules (Fig. 10). Moreover the transformation $\mathcal{T}_{\mathcal{MV}}(P)$ preserves the semantics of the original program P.

We write ε_ℓ for a sequence of events unobservable at level ℓ and τ_ℓ for the projection of trace τ at level ℓ.

Lemma 12 (Single output secrecy). *Let P' be a program and $P = \mathcal{T}_{\mathcal{MV}}(P')$. For all m_1, m_2, $\mathcal{E}_1, \mathcal{E}_2$, ℓ and Γ such that $m_1 \approx_\Gamma m_2$, $\mathcal{E}_1 \approx_\ell \mathcal{E}_2$ and $\Gamma, pc, \mathcal{E}_1 \vdash \langle P, m_1 \rangle \overset{\varepsilon_\ell}{\twoheadrightarrow}{}^* \Gamma_1'', pc_1'', \mathcal{E}_1'' \vdash \langle P_1'', m_1'' \rangle \overset{\alpha_1}{\twoheadrightarrow} \Gamma_1', pc_1', \mathcal{E}_1' \vdash \langle P_1', m_1' \rangle$, there exist $P_2', \Gamma_2', \mathcal{E}_2', m_2'$ such that either $\Gamma_2, pc_2, \mathcal{E}_2 \vdash \langle P_2, m_2 \rangle \overset{\tau^*}{\twoheadrightarrow}{}^*$ and $\tau_\ell^* = \varepsilon$, or $\Gamma, pc, \mathcal{E}_2 \vdash \langle P, m_2 \rangle \overset{\varepsilon_\ell}{\twoheadrightarrow}{}^* \Gamma_2'', pc_2'', \mathcal{E}_2'' \vdash \langle P_2'', m_2'' \rangle \overset{\alpha_2}{\twoheadrightarrow} \Gamma_2', pc_2', \mathcal{E}_2' \vdash \langle P_2', m_2' \rangle$ and $\alpha_1 \approx_\ell \alpha_2$, $m_1' \approx_{\Gamma_1'} m_2'$, $m_1'' \approx_{\Gamma_1''} m_2''$, $\Gamma_1' = \Gamma_2'$ and $\Gamma_1'' = \Gamma_2''$.*

Proof. We proceed by induction on the length of $\xrightarrow[ot]{\varepsilon_\ell}\!\!\!\!\twoheadrightarrow^*$ and P, leveraging the instrumented semantics for observable tracking in Fig. 7 and the transformation $\mathcal{T}_{\mathcal{MV}}(P')$.

Lemma 13. *Let P' be a program and $P = \mathcal{T}_{\mathcal{MV}}(P')$. For all m_1, m_2, $\mathcal{E}_1, \mathcal{E}_2$, ℓ and Γ such that $m_1 \approx_\Gamma m_2$, $\mathcal{E}_1 \approx_\ell \mathcal{E}_2$ and $\Gamma, pc, \mathcal{E}_1 \vdash \langle P, m_1 \rangle \overset{\tau_1}{\twoheadrightarrow}{}^* \Gamma_1', pc_1', \mathcal{E}_1' \vdash \langle P_1', m_1' \rangle$, there exist $P_2', \Gamma_2', \mathcal{E}_2', m_2'$ such that $\Gamma, pc, \mathcal{E}_2 \vdash \langle P, m_2 \rangle \overset{\tau_2}{\twoheadrightarrow}{}^* \Gamma_2', pc_2', \mathcal{E}_2' \vdash \langle P_2', m_2' \rangle$ and either $\tau_1 \approx_\ell \tau_2$, or $\tau_{1\ell}$ is a prefix of $\tau_{2\ell}$ and $\Gamma_2', pc_2', \mathcal{E}_2' \vdash \langle P_2', m_2' \rangle \overset{\tau^*}{\twoheadrightarrow}{}^*$ and $\tau_\ell^* = \varepsilon$.*

Proof. Induction on sequence of events at level ℓ and Lemma 12.

Lemma 14. *For any program P and initial state $(\mathcal{E}, m)$, $OS \vdash_{\mathcal{E},m} \mathcal{T}_{\mathcal{MV}}(P) \Rightarrow FS \vDash_{\mathcal{E},m} P$*

Proof. Follows from Lemma 13 and Lemma 11.

Proof (Proof for Thrm 7).
Follows from Lemma 14 and Lemma 8.

E Advanced Features and Implementation

In this section, we discuss how to bring the staged analysis presented previously into line with our implementation. We extend the language in Fig. 2 to a realistic language with heap locations, objects and arrays, and discuss challenges related to soundness and precision of the analysis. We also show how to incorporate notions of deliberate release of sensitive information, e.g. *declassification*, which very often appear in real-world applications. Finally, we discuss our implementation for the TaintDroid weak tracker and the intermediate language used to implement the transformations.

E.1 Language Extensions

The language from Section 2.1 can be extended with heap locations, objects, arrays, exceptions and method calls. Fig. 12 describes the full grammar of the language handled by our implementation. We briefly show how to extend the semantics, and discuss related issues.

We extend program states and configurations with heap $h \in \mathcal{H}$, mapping locations and field identifiers to values, i.e. $\mathcal{H} = Loc \times Fld \to Val$. The set of values Val is extended with object references $l \in Loc$ and special values **null** and **void**. An object $Obj : Fld \to Val$ is a total function from object fields $Fld \subseteq Var$ to values. The heap $\mathcal{H} : Loc \hookrightarrow Obj$ is a partial function from non-null object references $Loc \subseteq Val$ to objects and it is extended as expected whenever a new object is created.

The instrumented semantics in Fig. 7 is given by extending the security environment with labels for object references, namely $\Gamma : Var \cup Loc \mapsto \mathcal{L}$. Assigning labels to object references is needed as programs can leak sensitive information through object aliasing, as shown by the following example:

$$\begin{array}{l} A\ a_1 = \textbf{new}\ A(), a_2 = \textbf{new}\ A() \\ \textbf{if}\ h\ \textbf{then}\ a_1 = a_2\ \textbf{else skip} \\ \textbf{out}(L, a_1 == a_2) \end{array}$$

If object references do not have a security label, then this program will bypass the enforcement mechanism despite leaking the value of h through an observable implicit flow. We discuss this problem further in the context of our implementation in Section E.3.

Arrays: To maintain soundness, arrays require both security labels for each element as well as a label for the entire array. This is also matched by the behavior of practical taint tracking systems such as TaintDroid [24].

Method Calls: It is worth to point out that this technique requires no special handling of method calls; all that is required to track the security level of the *pc* globally, since methods may be called in a high context and perform public outputs there.

Reflection: Since reflection basically represents a branch in the control flow, our techniques can be extended to provide support for reflection by raising the level of the *pc* variable to the reflexive expression; e.g. when encountering an expression like $\textbf{eval}(e)$, we add an instruction $pc := e \oplus \iota(pc)$ to take into account that e influenced what code has been executed.

Exceptions: Since our technique assumes access to the control-flow graph of the program, exceptions can be handled similarly to other branching points in a program's control flow. Whenever encountering a statement that may result in an exception, we raise the level of *pc* to the level of this expression. This will prevent leaks in programs such as $\textbf{try}\ x := \frac{1}{h}\ \textbf{catch}(e)\ \textbf{out}(L, 1)$.

If exceptions are not actually caught and instead would result in abnormal termination, no extra action is required as we target progress-insensitive secrecy.

E.2 Declassification

The security policies presented in Sect. 2 are often too strict to accommodate real-world applications. For example, sending the result of an encrypted high value over a low channel breaks the security conditions, however, such operation is usually allowed in practice. We show how to extend our approach declassify data that depends on sensitive information.

We extend the language with a new construct, $\textbf{declassify}(e)$, that has the same semantics as expression e, and it is only used for intentional release of the value of e at a give program point.

Declassification policies appear in different flavors in the literature [45]. The enforcement mechanisms in Sect. 3 can be extended to transparently account for the *What* and the *Where* dimensions. Consider the following program branching on a high variable h and declassifying the value of another high variable h' in the true branch of the conditional.

$$\begin{array}{l}\textbf{if } h \textbf{ then } l := \textbf{declassify}(h') \textbf{ else skip}\\ \textbf{out}(\mathbf{L}, l)\end{array}$$

The program is insecure since an attacker who observes the value of variable h' (allowed by the declassification policy), is able to deduce information about the value of variable h, which they are not allowed to. To prevent similar types of leaks, the enforcement mechanism disallows occurrences of **declassify**() statements in high security contexts. Interestingly, the transformations for observable and full tracking remain unchanged; the only requirement is that the underlying weak tracker have support for declassification statements. This strategy corresponds to enforcing the *Where* dimension, which may be subject to *laundering* attacks, as in the following example.

$$h' := h \; ; \; l := \textbf{declassify}(h') \; ; \; \textbf{out}(\mathbf{L}, l)$$

The program launders the value of secret variable h through declassification of variable h'. We can account for such leaks by ensuring that variables appearing in declassification expressions are never modified prior to reaching the expression. This strategy would rule out the program above as h' is modified before reaching the **declassify**(h') expression. Again, this is an issue that needs to be handled at the level of the weak tracker, therefore the transformations for observable and full secrecy remain unchanged.

Soundness of mechanisms with declassification support can be shown using knowledge-based conditions for declassification, e.g *gradual release* [4] or *delimited release* [46].

E.3 TaintDroid

TaintDroid lacks built-in support for declassification. To work around this limitation, we implement declassification of tainted values by leaking them bit-wise through the control. The code transformation then ignores specifically listed declassification methods to avoid tracking intended implicit leaks. Moreover, TaintDroid only stores taint status for primitive values, but not for objects. This can result in leaks through aliasing as discussed in Appendix E.1. While extending TaintDroid to store taint status for objects is beyond the scope of this article, a possible workaround is to store each object in a singleton array and replace each usage of an object by accessing the first element of that array. This will prevent such leakage as TaintDroid stores the taint status of each array, while impacting performance and code size.

As is common for Android-related implementations, we do not address native code. Moreover, we assume that the sources and sinks, though not necessarily their security levels, are known at time of transformation.

E.4 Intermediate Language

Our tool is a prototype built on top of the *Soot* framework [56] and it uses an intermediate bytecode language, *Jimple* [56], to implement the static transformations presented in Sect. 3.

Jimple is a typed and stackless 3-address intermediate representation language for Java bytecode. We present the syntax for the constructs of Jimple in Fig. 12. Jimple programs consist of a list of class declarations, and a *main* method that serves as unique entry point to the program. Classes are lists of instance fields and methods. Constructors are also represented as methods. Static fields and static methods are similar (not reported here). Each method consists of a *signature* and a list of instructions. Formal parameters and reference *this* are represented at the beginning of each method as assignments to local variables for all the formal parameters and for the current class instance, respectively.

$$
\begin{aligned}
e ::=\ & n \mid x \mid e \oplus e \mid \textbf{new}\ C \mid e.f \mid \textbf{newarray}\ (T)\ [e] \\
& \mid e[e] \mid \textbf{length}\ x \mid \textbf{specialinvoke}\ x.m(\overline{e}) \\
& \mid \textbf{virtualinvoke}\ x.m(\overline{e}) \\
c ::=\ & \textbf{nop} \mid x := e \mid x.f := e \mid x[e] := e \mid \textbf{out}(\ell, e) \mid \textbf{goto}\ pc \\
& \textbf{if}\ e\ \textbf{goto}\ pc \mid \textbf{invoke}\ e \mid \textbf{return}\ e \mid \textbf{throw}\ e \\
M ::=\ & A\ m(\overline{x})\ \{\ \overline{c}\ \} \\
C ::=\ & \textbf{class}\ A\ \{\overline{A\ f};\ \overline{M}\} \\
P ::=\ & \overline{C}
\end{aligned}
$$

Fig. 12. Grammar of Jimple

Moreover, methods end with a **return** e instruction to denote a possible return value. Jimple contains assignments to local variables, fields and array locations. It uses a 3-address code representation, meaning that at most two operands can occur on the right-hand side of an assignment. Conditional and unconditional jumps are used to transfer the control flow to a given code location. Jimple generalizes conditional jumps to several branches using a *switch* statement (not shown here). Invocation statements use **invoke** to call a constructor (via **specialinvoke** expression) or a method (via **virtualinvoke** expression). **invoke** ignores the possible result of a method call. **throw** e throws an exception specified by the class of local variable e. We represent API sinks by output statements $\textbf{out}(\ell, e)$. Jimple expressions are defined as expected; n denotes a value which is an integer, a reference, **null** or **void**. x denotes a local variable of either primitive or object type. $\oplus$ denotes a binary operation (e.g. $+, -$) or a binary relation (e.g. $<, \&$). **new** and **newarray** denote, respectively, the creation of an object reference and of an array of respective type and size, while **length** e denotes the length of the array referenced by e. We refer the reader to [56] for the full syntax and semantics of Jimple.