Explicit Secrecy: A Policy for Taint Tracking

Daniel Schoepe*, Musard Balliu*, Benjamin C. Pierce†, and Andrei Sabelfeld*
*Chalmers University of Technology
†University of Pennsylvania

***Abstract*—*Taint tracking* is a popular security mechanism for tracking data-flow dependencies, both in high-level languages and at the machine code level. But despite the many taint trackers in practical use, the question of what, exactly, tainting *means*—what security policy it embodies—remains largely unexplored.**

We propose *explicit secrecy*, a generic framework capturing the essence of explicit flows, i.e., the data flows tracked by tainting. The framework is semantic, generalizing previous syntactic approaches to formulating soundness criteria of tainting. We demonstrate the usefulness of the framework by instantiating it with both a simple high-level imperative language and an idealized RISC machine. To further understanding of what is achieved by taint tracking tools, both dynamic and static, we obtain soundness results with respect to explicit secrecy for the tainting engine cores of a collection of popular dynamic and static taint trackers.

1. Introduction

Taint tracking is a striking success story in computer security. It is used to enhance both confidentiality and integrity in a wide variety of applications ranging from hardware-level [23], [56] and binary-level tainting [22], [54] to tainting in mobile [34] and web [46], [59], [43] applications. Languages such as Perl [4] and Ruby [3] have built-in support for taint tracking, and extensions for languages such as Java [37], [59], JavaScript [46], and Python [28] are available to perform taint tracking.

Motivation. At its heart, taint tracking is about tracking data dependencies as data is propagated by the system. In the setting of a mobile app, taint tracking can help detect privacy leaks, e.g., when the app attempts to send the user's location to a third-party. This is done by labeling location data as secret and detecting when secret-labeled data is sent to third parties over the network, as e.g. in the popular TaintDroid tool [34]. This is an example of enforcing a confidentiality policy by taint tracking. Or, in the setting of a web application, some input sources (e.g., user input) can be labeled as "tainted," and taint tracking can be used to prevent tainted data from affecting sensitive sinks (e.g., writing to a system file or generating HTML for a web page) [59], [43]. This is an example of enforcing an integrity policy by taint tracking.

A key reason behind the success of taint tracking tools in practice is that taint tracking is a pure data dependency analysis. It only tracks *explicit* [32] flows of the form $l := h+1$ where the value of h is explicitly leaked into l, while ignoring *implicit* [32] flows of the form **if** h **then** $l := 1$ **else** $l := 0$ via the control-flow structure of the program. Ignoring implicit flows makes taint trackers unsound (especially for malicious code where the attacker is in control of what flows to exploit [50]), but this loss of soundness is compensated by a large increase in practicality because the enforcement need not track control flows. This makes a crucial difference for the practicality because tracking control flow is hard: although much progress has been made on information-flow tracking, dealing with control flow in expressive programming languages remains challenging [51], especially in dynamic languages as JavaScript where it is hard to predict side effects along alternative execution paths [38]. By comparison, taint tracking is appealingly lightweight and effective, as evidenced by its success in finding vulnerabilities in real systems and its adoption in many programming languages.

The success of taint tracking brings us to a seemingly basic yet highly elusive question: What, precisely, is taint tracking good for? In other words, what is the formal meaning of "data dependency" and "explicit flows," and what is an appropriate soundness criterion for a taint tracker? Answers to these questions are not to be found in program analysis literature [48], where the focus is not on data dependencies themselves but on their use for code transformation and optimization [7]. What is more surprising is that general answers to these basic questions, to the best of our knowledge, are neither to be found in the security literature!

This is in sharp contrast to the general area of *information flow* [51], which is concerned with specifying confidentiality and integrity for both explicit and implicit flows, where there is a large body of work on policies ranging from *noninterference* [27], [35], which allows no dependencies from secret sources to public sinks, to various flavors of *declassification* [52] and *endorsement* [11] policies.

It is thus highly desirable to improve the understanding of explicit (vs. implicit) flows and develop a general policy framework for taint tracking that can be related to known security characterizations and serve as a target condition for taint tracking mechanisms.

Background. To date, efforts on characterizing tainting policies have been scarce. Rather than specifying a security condition, it is common to state desired properties of the

enforcement, phrased in an enforcement-centric fashion, as, e.g., in graph-based properties in the work by Livshits and Chong [42], [43]. Another approach is to formalize the essence of taint tracking by formalizing a generic taint tracker, as, e.g., in the work by Schwartz et al. [54]. While succinctly representing what happens inside such tools as BitBlaze [56] and BAP [19], such a formalization is inherently low-level [54].

What we are looking for is an *enforcement-independent* condition that captures the essence of explicit flows and that can be checked against independently interesting approaches to enforcement.

Closest to our needs is Volpano's *weak secrecy* [63], the only policy we are aware of that focuses on describing what can be enforced by explicit flow analysis. Weak secrecy makes use of the classical information-flow notion of *noninterference* [27], [35], which demands that a program's secret input may not influence the program's public output. Intuitively, a program satisfies weak secrecy if, for any run of a program up to a given point in time, the sequence of assignments performed by the program satisfies noninterference.

While weak secrecy is a reasonable starting point, there are some roadblocks for adopting it for reasoning about practical taint trackers. The fact that the definition is *syntactic* in nature, relying on extracting assignment commands from the original program, implies that the definition does not scale to languages with rich features such as reflection. This makes adapting it to low-level languages particularly challenging. Low-level machines allow "reflective" programming idioms such as programs that read or write their own instructions. A direct extension of weak secrecy in this setting does not work because a modified program will not necessarily have the same instructions in the memory. Also, low-level machines typically have many instructions, and each instruction may have complex semantics (e.g., jump-and-link instructions that modify both control flow and memory). There is no clear criterion for how to extend the weak secrecy definition. This is particularly concerning, given that taint tracking often targets low-level machines [22], [54]. Other features that challenge weak secrecy, even in high-level languages, include expressions with side effects, which require custom-tailored encodings to get weak secrecy to work. Finally, the definition is indirect, defining a weak policy, weak secrecy, via the stronger policy of noninterference.

What we want is a *language-independent*, *semantic* definition of "correct taint tracking", generalizing weak secrecy and applicable to a wide range of models from high-level languages to low-level machines. Our goal is to lay foundation for exploring the design space of ways to split program configurations into data and control.

Contributions. Motivated by the above, we propose a general semantic framework for specifying explicit flows, offering the following contributions: (i) We propose a knowledge-based semantic security condition, *explicit secrecy*, that captures the essence of explicit flows in a language-independent way, based on a distinction between "control" and "data" that is specified by the language designer. Intuitively, explicit secrecy separates data and control and demands the security of flows for the data part. (ii) We show the flexibility of the model by incorporating possibilities of *declassification* and *sanitization*, unexplored features in the context of previous attempts to give a soundness condition to taint tracking. (iii) We instantiate explicit secrecy for a high-level imperative language with I/O to obtain a soundness criterion for taint tracking. (iv) We instantiate explicit secrecy for a simple RISC machine to yield a soundness criterion in the setting of low-level languages. (v) Being a semantic condition, explicit security can readily be related to known security characterizations. We show that its instantiation to a simple imperative language agrees with weak secrecy, demonstrating that explicit secrecy indeed generalizes weak secrecy. Thus, like weak secrecy, explicit secrecy does not subsume (and is not subsumed by) noninterference. Further, we establish that explicit secrecy is stronger than an intuitive declassification-based condition (dubbed Control-Flow Gradual Release) that declassifies the guards at branching points. (vi) We use explicit secrecy to illuminate the behavior of real-world taint trackers by analyzing the core of taint tracking engines from several popular languages and tools. In particular: we show soundness for a dynamic enforcement mechanism for a simple subset of high-level languages such as Perl and Ruby; we show soundness for a dynamic enforcement mechanism [54] underlying the BAP [19] and BitBlaze [56] tools for low-level machines; and we define a simple static enforcement mechanism reminiscent of the ones in Andromeda [60] and FlowDroid [8] and prove it sound.

Scope. The paper lays groundwork for judging the soundness of existing and new enforcement mechanisms. We evaluate the foundational framework on the core mechanisms underlying practical enforcement techniques. The approach gives benefit whether the soundness proofs succeed (by giving assurance of what is achieved) or fail (by pointing to insecurities). Thus, in addition to the foundational impact, the practical impact of the paper is the demonstration that core mechanisms of the practical tools (analyzed in Section 3) are sound. While experimental evaluation of the approach is important, it can only be done once a sound foundation is in place. Similarly to other foundational work on security (e.g., [44], [12], [25], [54]), we note that while the experimental evaluation is not in the scope of the present study, it is subject to subsequent engineering work that can build on the foundations.

For simplicity, we focus on confidentiality in the rest of the paper. Note that noninterference is suitable for both integrity (e.g., tracking buffer overruns) and confidentiality (e.g., tracking leaks in mobile apps). These are dual properties in terms of information flow [17]. Similarly, our notion of explicit secrecy is equally suitable for both integrity and confidentiality, through dualization. Using the duality, we can interpret tainted sources/sinks as untrusted and reason about attacker influence similarly to knowledge-based approaches. Sanitization functions for preventing injection vul-

nerabilities can be modeled as declassification/endorsement.

For brevity, we elide proofs of formal statements and full details of the larger definitions; these are available in the full version of the paper, available online [6].

2. Specifying Explicit Flows

We first review the definition of *weak secrecy* and elaborate on difficulties with instantiating weak secrecy to richer languages (Section 2.1). To address these difficulties, we introduce *explicit secrecy* (Section 2.2), a language-agnostic security property that formalizes the idea of "security with respect to explicit flows only." We show how to instantiate this property to both a simple imperative language (Section 2.4.1) and machine code for a simple RISC machine (Section 2.4.2). To benchmark explicit secrecy against information-flow conditions, we show examples demonstrating that it is incomparable in power to noninterference. Further, we explore an intuitive approach to characterize explicit-flow tracking using a variant of *gradual release* [12], a knowledge-based information release policy. We show that explicit secrecy is stronger than this characterization (Section 2.5).

2.1. Weak Secrecy

To formalize security with respect to (just) explicit flows, Volpano [63] introduced *weak secrecy* for a simple imperative language. We begin by recapitulating his definition.

Our language includes global variables, while loops, conditionals, assignment, and output.

$$c ::= \ \textbf{skip} \mid c_1; c_2 \mid x := e \mid \textbf{out}\ e \mid \\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ e\ \textbf{do}\ c$$

We assume a set of variables Var. A configuration (c, m) consists of a command c and a memory $m \in Mem$ mapping variables to integers, i.e. $Mem = Var \to \mathbb{Z}$. We write $(c, m) \xrightarrow{\alpha} (c', m')$ to denote that a configuration (c, m) evaluates in one step to configuration (c', m') while producing trace $\alpha \in Obs^*$, where $Obs = \mathbb{Z}$. The (standard) definition of this relation can be found in the extended version. A terminated program is represented by ε. We also write ε for the empty trace.

Clark and Hunt [24] show that for the security of deterministic programs, such as programs in the presented language, it makes no difference whether the environments are modeled as streams or strategies. A further common simplification [10] is to model input to a program by initial memories.

We assume that each variable x has an associated security level $\Gamma(x) \in \mathcal{L}$, where $(\mathcal{L}, \sqcup, \sqsubseteq)$ is a lattice of security levels. In the examples, we assume a two-level security lattice consisting of **H** for variables containing confidential information and **L** for variables containing public information, with $\mathbf{L} \sqsubseteq \mathbf{H}$. Two memories m_1 and m_2 are said to be *low equivalent*, written $m_1 =_{\mathbf{L}} m_2$, iff $\forall x.\ \Gamma(x) = \mathbf{L} \Rightarrow m_1(x) = m_2(x)$.

Our language differs in one small respect from the one used by Volpano: there, every assignment to a low variable generates an event that is visible to the attacker. While this allows for content in high variables to be updated with low values, any assignment of high content to low variables renders the program insecure. We make the setup a bit more flexible by limiting the attacker to observing the program's external behavior, introducing explicit output instructions that generate attacker-visible events, with assignments not being directly observable. However, this choice is orthogonal to the security conditions presented here and in Section 2.2.

The intuition of weak secrecy is that every possible sequence of non-control-flow statements (assignments and outputs) executed by a program run has to be noninterfering. To define this, we annotate the evaluation rules in the semantics to record executed explicit flow statements, writing $(c, m) \xrightarrow[d]{\alpha} (c', m')$ if the step from (c, m) to (c', m') generates the explicit flow statement d (and observable events α). Note that d is distinct from α—i.e., it is not part of the trace of attacker-visible observations generated by the program. For example, here are the instrumented rules for assignment, output, and the true case of the conditional[1] (the extended version gives the other rules); when d and/or α are empty or unimportant, we omit the superscript and/or subscript on the $\to$:

$$\frac{m(e) = n}{(x := e, m) \xrightarrow[x:=e]{} (\varepsilon, m[x \mapsto n])} \quad \text{(E-ASSIGN)}$$

$$\frac{m(e) = n}{(\textbf{out}\ e, m) \xrightarrow[\textbf{out}\ e]{n} (\varepsilon, m)} \quad \text{(E-OUT)}$$

$$\frac{m(e = 0) = n \qquad n = 0}{(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, m) \to (c_1, m)} \quad \text{(E-IFTRUE)}$$

Note that the notion of "control-flow statement" is not a formal one and is left open when instantiating weak secrecy to another language. While for this language it is reasonably straight-forward to determine which statements affect the memory, more advanced language features can blur this issue.

We use a standard definition of noninterference [27], [35]. Because we will only apply noninterference to straight-line programs the exact flavor (e.g., termination-sensitive vs. insensitive [51]) is inconsequential.

2.1.1 Definition [Noninterference]: A program c satisfies *noninterference*, written $NI \vDash c$, iff whenever $m_1 =_{\mathbf{L}} m_2$ and $(c, m_1) \xrightarrow{\ell_1}{}^* (\varepsilon, m_1')$ and $(c, m_2) \xrightarrow{\ell_2}{}^* (\varepsilon, m_2')$, we have $\ell_1 = \ell_2$.

Finally, we define weak secrecy in terms of the extracted explicit flow statements from all possible runs:

1. We assume that an expression such as $e = 0$ evaluates to 1 if $m(e) = 0$ and 0 otherwise.

2.1.2 Definition [Weak secrecy]: A program c satisfies *weak secrecy*, written $WS \vDash c$, iff whenever $(c, m) \xrightarrow[d]{}{}^{*} (c', m')$, we have $NI \vDash d$.

Consider the following program (call it c_{impl}):

```
if (h = 0)
  then l := 1
  else l := 2;
out l
```

This program leaks whether or not the high variable h is 0 using an *implicit flow*. For c to satisfy weak secrecy, all possible sequences of assignment and output statements arising from executions of this program in different starting memories must be noninterfering. Concretely, $l := 1$ and $l := 2$, as well as $l := 1$; **out** l and $l := 2$; **out** l have to satisfy noninterference. Since this is the case, $WS \vDash c$ holds. This captures the intuition that taint is not propagated through control-flow decisions in taint-tracking systems.

Weak secrecy offers an intuitive formal account of security with respect to explicit flows. However, its inherently syntactic nature ("extract the assignment statements and outputs from every run of the program...") means that it is not always clear how to extend it to other programming languages. For example: (i) Many high-level languages allow the expressions appearing in the guards of conditionals and loops to have side effects. In such languages, the extracted statement will have to include more than just top-level assignment statements and outputs from the original program. E.g., for a program such as **if** $(h := 0) = 0$ **then** ... **else** ..., the assignment $h := 0$ has to be extracted. (ii) Conversely, many languages allow conditional *expressions* such as $x \ ? \ y : z$ in C. If such expressions are regarded as "control flow," then some significant transformation may be required to extract just the data-flow parts of expressions. Weak secrecy gives no direct guidance as to how this should be done.

One particular setting where taint tracking is commonly employed in practice is machine code. But adapting weak secrecy to machine code again raises questions: (i) Machine-code programs can access their own "program text" as data (and they sometimes do this, in practice, for example in certain tamper-proofing techniques [45]). If we extract only a part of this program text into the extracted program, then program memory will contain different *values* when the extracted program is executed, compared to the original. The extracted program can then produce different results, resulting in insecure flows that only occur in the extracted program but not in the original, or vice versa. We discuss this further in Section 2.4.2. (ii) Some instructions, e.g. jal (jump and link), modify both the machine's memory (the stack) and its control state (the program counter); that is, there is both an explicit as well as an implicit flow of information. To extract only the explicit flow part, it seems jal instructions would have to be extracted into instructions capturing only the changes to the memory.

2.2. Explicit Secrecy

To deal uniformly with all these issues, we introduce *explicit secrecy*, a semantic property that builds on the intuition of weak secrecy. Intuitively, explicit secrecy separates changes to the state of a program from changes to the control flow, like weak secrecy. However, rather than manipulating the syntax of the running program, we now extract a *function* that captures the "state modification" of each execution step.

Explicit secrecy is language-agnostic, assuming only that the language designer can give a (small-step) semantics for their language or machine and specify the distinction between the "state" and "control" parts of a program or machine configuration with this choice determining what is considered to be an explicit flow. Roughly speaking, choosing a larger part of a configuration as the "state" part entails considering more flows as explicit flows. Concretely, the interface between the language definition and the explicit secrecy property comprises the following:

i. Sets of configurations ($Conf$), commands (Com), states ($State$), and observations (Obs). Intuitively, a state denotes only the "memory part" (as opposed to the "control part," which is a command) of a configuration. An observation is an event that is visible to the attacker. (We sometimes say "program" instead of "command" when we want to emphasize that we are talking about an entire program such as might be loaded into a machine at the beginning of a run, as opposed to smaller fragments such as single machine instructions.)

ii. A small-step evaluation relation: $\rightarrow \subseteq Conf \times Obs^* \times Conf$. We write $cfg \xrightarrow{\alpha} cfg'$ if cfg evaluates to cfg' in one step while producing observations α; we elide α when it is empty or unimportant. For simplicity, we assume that this relation is deterministic. (We discuss how to lift this assumption later in this section.)

iii. An equivalence relation $=_{\mathbf{L}} \subseteq State \times State$ between states, where $s_1 =_{\mathbf{L}} s_2$ means that the states s_1 and s_2 are not distinguishable by the attacker.

iv. A function $state(\cdot) : Conf \rightarrow State$ mapping configurations to states and a function $com(\cdot) : Conf \rightarrow Com$ that extracts the *next* command that will be executed. Moreover, we require a function $\langle \cdot, \cdot \rangle : Com \times State \rightarrow Conf$ constructing a configuration with a given command and state. For every configuration cfg it must hold that $\langle com(cfg), state(cfg) \rangle = cfg$.
(In the instantiation for the while language in Section 2.4.1, configurations are simply tuples of command and state parts; $\langle \cdot, \cdot \rangle$ constructs a tuple, with $state(\cdot)$ and $com(\cdot)$ acting as projections and obeying the usual laws. In the case of machine code, however, data and code are mixed. Therefore, the command that is extracted consists only of the next instruction, since other memory content cannot be distinguished from data that will not be executed. Thus, $com(\langle c, s \rangle) = c$ does not hold in general for machine code. We discuss this in more detail in Section 2.4.2.)

The power of explicit secrecy depends on careful choice of the parameters. Intuitively, explicit flows are only concerned

with changes to the state. To model explicit flows, we use the small-step semantics to construct, for each execution step during a program's execution, a *function* that transforms a state while possibly producing output events. For simplicity, we add another assumption to guarantee totality of the function constructed for each step: we assume that, if $cfg \rightarrow cfg'$, then for every configuration cfg_1 with $com(cfg) = com(cfg_1)$, there exists cfg_1' such that $cfg_1 \rightarrow cfg_1'$. I.e., evaluation of a fixed command is defined for all possible states, if that command can be evaluated in *some* state. This, together with assuming a deterministic evaluation relation, allows us to extract functions acting on state in a unique and well-defined way.

2.2.1 Definition: For each step $cfg \rightarrow cfg'$, we define a function $f : State \rightarrow State \times Obs^*$ by $f(s) = (state(cfg''), \alpha)$ for the unique α, cfg'' such that $\langle com(cfg), s\rangle \xrightarrow{\alpha} cfg''$. We write $cfg \underset{f}{\rightarrow} cfg'$ to denote that cfg evaluating to cfg' is associated with f in this way.

(Whether the subscript on $\rightarrow$ refers to a function, or an extracted command in the sense of weak secrecy, should always be clear from context.)

Intuitively, for a step $cfg \underset{f}{\rightarrow} cfg'$, $f(s)$ simulates how the given input state s would be changed by executing the same step that cfg performs. For example, executing an assignment $x := e$ in the while language results in $f(m) = (m[x \mapsto m(e)], \varepsilon)$, where $m(e)$ denotes evaluating expression e in memory m; i.e., executing an assignment modifies the memory accordingly while producing no output. Performing a step for a command of the form **if** e **then** c_1 **else** c_2 results in $f(m) = (m, \varepsilon)$, encoding the fact that branching neither changes the memory nor produces any output.

We can then lift the construction of state changing functions to multiple steps by chaining the state modifications and concatenating the produced output

$$\frac{}{cfg \underset{id}{\longrightarrow}^* cfg} \qquad \frac{cfg \xrightarrow[f]{\alpha}{}^* cfg' \qquad cfg' \xrightarrow[g]{\beta} cfg''}{cfg \xrightarrow[g \odot f]{\alpha.\beta}{}^* cfg''}$$

where $g \odot f$ computes the final state and concatenates the outputs—i.e., $(g \odot f)(s) = (s'', \alpha.\beta)$, when $f(s) = (s', \alpha)$ and $g(s') = (s'', \beta)$.

For example, consider the command $c = (h := 0$; **out** $h)$ in the simple imperative language of Section 2.1. The function g associated with executing two steps of this program, i.e., $(c, m) \xrightarrow[g]{\ell}{}^* (\varepsilon, m')$, will first perform update the current memory and then output the value of h in the modified memory. That is, $g = g_2 \odot g_1$ where $g_1(m) = (m[h \mapsto 0], \varepsilon)$ and $g_2(m) = (m, [m(h)])$, and hence $g(m) = (m[h \mapsto 0], [0])$.

Functions constructed this way correctly encode actual execution steps:

2.2.2 Lemma: If $cfg \xrightarrow[f]{\alpha}{}^* cfg'$, then

$$f(state(cfg)) = (state(cfg'), \alpha).$$

Depending on how programs and configurations are represented, not all initial states might be valid starting states for a program. For example, in the low-level language we consider, starting configurations are created from a program, represented as a list of instructions, and a starting memory by overwriting a part of the memory with the instructions of the program. Therefore, considering states where the program region contains different instructions can render well-behaved programs insecure. (A concrete example of this can be found in Section 2.4.2.) To rule out such "impossible" initial states, we define *valid initial states* as the set of states that can be obtained by creating a configuration from the command:

2.2.3 Definition [Initial states]: For command c, we define the set of *valid initial states* $\mathcal{S}_0(c)$ by

$$\mathcal{S}_0(c) = \{s \mid \exists s'.state(\langle c, s'\rangle) = s\}.$$

In the case of machine code programs, this set will contain memories that match the instructions of c at the addresses where c is placed. In the case of while programs, all states are valid initial states.

We can now define the knowledge an attacker obtains from observing only outputs from a sequence of changes to the state. We capture this by defining a set of initial states that the attacker considers possible based on some observations. Concretely, for a given initial state s_0 and some state transformer f, another state is considered possible if $s_0 =_{\mathbf{L}} s$ and it matches the trace produced by $f(s_0)$, i.e. $\pi_2(f(s_0)) = \pi_2(f(s))$, where π_i projects a tuple to its ith component.

2.2.4 Definition [Explicit knowledge]: We define the *explicit knowledge* with respect to command c, initial state s_0, and state transformer f by

$$k_e(c, s_0, f) = \{s \mid s =_{\mathbf{L}} s_0 \ \wedge \ s \in \mathcal{S}_0(c) \wedge \pi_2(f(s)) = \pi_2(f(s_0))\}.$$

A program then satisfies explicit secrecy for some initial state iff no indistinguishable, valid initial states can be ruled out from observing the output generated by the extracted state transformer.

2.2.5 Definition [Explicit secrecy]: A program c satisfies *explicit secrecy for initial state* s, written $ES \vDash (c, s)$, iff whenever $\langle c, s\rangle \underset{f}{\rightarrow}^* cfg'$, we have $\forall s_0 \in \mathcal{S}_0(c)$. $k_e(c, s_0, f_0) = k_e(c, s_0, f)$ where $f_0(s) = (s, \varepsilon)$. A program c satisfies *explicit secrecy*, written $ES \vDash c$, iff $ES \vDash (c, s)$ for all $s \in State$.

In order to ignore information from implicit flows, the definition quantifies over all $s_0 \in \mathcal{S}_0(c)$ instead of states that are low-equivalent to s. For example, consider the program **if** $l = 0$ **then out** $l \times h$ **else skip**, which is considered insecure by most taint-tracking systems. The state transformer extracted for the **then** branch (for a memory m_0 where $m_0(l) = 0$) is $f(m) = (m, [m(l) \times m(h)])$. If we consider only states that are low-equivalent to m_0, the explicit flow in the **then** branch is not detected, since low equivalence

to m_0 implies that $l = 0$. In order to capture the behavior of taint-tracking systems, we quantify over all (valid) initial states instead.

Since the definition is knowledge-based, it extends naturally to a non-deterministic setting by extending state transformers to return sets of possible successor states. Similarly, the totality assumption on the evaluation relation can be lifted by constructing partial state transformers and considering only states in the domain of the resulting state transformers.

2.3. Declassification

Like noninterference, explicit secrecy is often too strict to accommodate real-world applications and needs a mechanism to *declassify* some data that depends on sensitive information. For example, taint-tracking systems often allow taint to be removed from data after passing it through a sanitizer. In this section we show how explicit secrecy can be extended to handle such behavior.

Being a knowledge-based definition, explicit secrecy can be extended naturally to support declassification in the style of gradual release [12]. We assume that there is a set of release events $Rel \subseteq Obs$ that occur when information is intentionally released by a program. In terms of explicit secrecy, we allow the attacker's knowledge to change based on observing such events but require that it remain constant for other events.

2.3.1 Definition [Explicit secrecy modulo release]: A command c then satisfies *explicit secrecy modulo release* for initial state s iff

$$\langle c, s\rangle \underset{f}{\rightarrow}^* cfg' \xrightarrow[g]{\alpha}{}^* cfg'' \wedge \alpha \notin Rel \Rightarrow$$
$$k_e(c, s, f) = k_e(c, s, g \odot f)$$

For example, the while language from Section 2.1 can be extended with a **declassify**(e) statement that releases the value of expression e to the attacker. The semantics of the language is extended by the following rule:

$$\text{E-DECL}\quad \frac{m(e) = n}{(\textbf{declassify}(e), m) \xrightarrow{rel(n)} (\varepsilon, m)}$$

The rule denotes that evaluating a statement **declassify**(e) produces a release event $rel(n)$ containing value n. We then define the set of release events as $Rel = \{rel(n) \mid n \in \mathbb{Z}\}$.

To illustrate this extension, consider a program that processes some sensitive input (such as a POST request containing a password and other data) and logs information about requests to a log file. It is important that the user's password not be leaked to the log file. For example, the following program logs the request verbatim, thereby violating the intended policy (this is often mentioned as a secure coding guideline as well [2]):

```
output_to_log ("Request: " + request);
if (hash(password(request)) == stored_hash)
then output_to_user sensitive_data
else output_to_user "access denied";
```

(Here `password` extracts the submitted password and `hash` computes the hash of a password.) This program is rejected by explicit secrecy since private information, namely the contents of `request`, are output explicitly (to the log file).

If instead the request is sanitized by removing the password, the program is secure and it is accepted:

```
declassify(remove_password(request));
output_to_log ("Request: " + remove_password(request));
if (hash(password(request)) == stored_hash)
then output_to_user sensitive_data
else output_to_user "access denied";
```

Because of the use of declassification, this program satisfies explicit secrecy modulo release: The attacker learns the value of `remove_password(request)` when the `declassify` command is executed; this is permitted, since the produced event is a release event. The same value is then appended to the log, resulting in no increase in knowledge for the attacker. (The final output statement also depends on the user's password, but does so only through the control flow and hence is accepted by explicit secrecy.)

2.4. Instantiating Explicit Secrecy

This section demonstrates how to instantiate the explicit secrecy framework both for the high-level language introduced in Section 2.1 and for a RISC-style assembly language.

As described in Section 2.2, instantiating explicit secrecy first requires the language designer to define which parts of a configuration hold state and which determine the control flow of a program. This is specified by providing mapping functions from configurations to state and command parts respectively.

For example, in many imperative high-level languages, configurations often consist of the current statement, along with various forms of state, such as values of variables, the state of the heap and stack, exception handlers, etc. In such a case, the statement and exception handlers (since they are part of the control flow) can be considered the control-flow component and the variables and the heap and stack as the state component of a configuration. Also note that, in the extremes, choosing the entire configuration as control-flow component yields a vacuously true condition, while taking the entire configuration to be the state part results in a form of noninterference. For example, in a purely functional language without side effects, the entire configuration has to be considered as either just state or just control, showing that taint tracking for (purely) functional languages is not a very meaningful notion.

Second, explicit secrecy assumes a small-step evaluation relation provided by the language designer, which also determines the events that are observable by the attacker. To model a powerful attacker observing all writes to public memory locations, each step of an assignment statement

to a low variable would produce an attacker observation. Conversely, for an attacker who only observes specific output events, observations are only produced when evaluating distinguished output statements.

To summarize, the following steps are needed when instantiating explicit secrecy for a new language:

- Decide what part of a configuration contains data. Explicit secrecy then only tracks statements that directly modify this part. Changes to other parts of the configuration are not considered for security. A useful rule of thumb may be to choose all parts of a configuration that directly contain secrets. Additionally, the language designer has to specify a function constructing configurations from a command and a state part.
- Choose an appropriate small-step relation. In order to arrive at a sensible security condition, evaluation steps need to be small enough to separate changes to the state from changes to the control-flow. For example, a big-step relation results in a much stronger security condition, similar to noninterference.
- Choose the attacker observations. This depends on the application scenario and affects how strong the resulting security condition is.
- Depending on the attacker, one should also choose an equivalence relation on states encoding what the attacker can observe about initial states.

2.4.1. Instantiation for While Language. To instantiate explicit secrecy for the language introduced in Section 2.1, we define the parametric elements from Section 2.2 as follows:

i. The set of states is given by memories, i.e., functions mapping variables to values $State = Mem = Var \to \mathbb{Z}$, and Com is the set of statements defined in Section 2.1. A configuration is a pair of a command and a memory: $Conf = Com \times Mem$. An observation is a value $v \in \mathbb{Z}$.
ii. The evaluation relation $\to$ is as given in Section 2.1.
iii. As before, $m_1 =_{\mathbf{L}} m_2$ holds iff $\forall x.\ \Gamma(x) = \mathbf{L} \Rightarrow m_1(x) = m_2(x)$.
iv. Projecting to the state or command component of a configuration extracts the memory $state((c, m)) = m$ or command part $com((c, m)) = c$; creating a configuration from a given command and memory is given by $\langle c, m\rangle = (c, m)$.

Each step results in a state transformer based on the first statement in the configuration. The state transformer acts on the memory part of the configuration, and might produce an output: An assignment $x := e$ will yield a function $f(m) = (m[x \mapsto m(e)], \varepsilon)$. Output statements of the form **out** e result in a function $f(m) = (m, [m(e)])$. For every other type of statement, the extracted function is $f(m) = (m, \varepsilon)$. In particular this means that control-flow commands such as **if** statements will not affect the attacker's knowledge, corresponding to the intuition that these are not considered explicit flows.

As an example, consider the program c_{impl} from Section 2.1, which leaks one bit about the value of h using an implicit flow. The possible state transformers f for a complete execution of c_{impl} starting in m_0 have the form $f_i(m) = (m[l \mapsto i], [i])$ where $i = 1$ or $i = 2$, depending on whether or not $m_0(h) = 0$. If we consider any other memory m such that $m \in [s_0]_{\mathbf{L}}$, then it holds that $\pi_2(f_i(m)) = [i] = \pi_2(f_i(m_0))$. Therefore, the program is judged secure by explicit secrecy, since no information is leaked through an explicit flow.

However, a program such as **if** $h = 0$ **then out** h **else out** 0 will be judged as insecure since, if $m_0(h) = 0$, the extracted state transformer is $f(m) = (m, [m(h)])$, due to the elimination of control-flow information. Even though this program is secure in the sense of noninterference, the program is not secure in the sense of explicit secrecy, since information is propagated using an explicit flow.

After performing a series of steps $(c, m) \underset{f}{\to}^* (c', m')$, the function f that is extracted from the run corresponds to the sequence of assignments and output statements executed; since the steps that only alter the control flow, such as **if** statements, do not change the memory, these statements are not reflected in f. This corresponds to the sequence of commands that is extracted by Weak Secrecy and in fact, weak secrecy and explicit secrecy for while programs coincide:

2.4.1 Theorem: $ES \vDash c$ iff $WS \vDash c$, for any command c.

A natural question that arises is how explicit secrecy applies to languages with richer features, such as pointers or reflection. For example, in a language supporting pointers, the extracted state transformers, depending on the exact semantics, would replicate the changes pointer-based features make to the memory. A statement like $*e_1 := e_2$ in a C-like language assigning the result of e_2 to the location pointed to by e_1 would result in the expected state transformer f: Namely $f(m)$ would evaluate e_1 in m, and assign the result of evaluating e_2 in m to that location and return the modified memory. This captures the state modification performed by the original statement. This would imply that an enforcement mechanism which taints all variables occurring in e_1 would be sound, provided the rest of the language features are handled properly. This is discussed in more detail in the context of machine code in the following section.

Similarly, features like Java-style reflection would result in state transformers capturing the modifications they perform to the stack and heap, but ignore implicit flows resulting from it; for example code constructed at runtime.

2.4.2. Instantiation for Machine Code. We next demonstrate how to instantiate explicit secrecy for a simple RISC-style machine. We denote the set of machine words by $\mathbb{W}$ and the set of registers by Reg. We consider the following instruction set:

$$
\begin{aligned}
i ::= \ & \textbf{nop} \mid \textbf{halt} \mid \textbf{out}\ r \mid \textbf{const}\ i\ r \mid \textbf{mov}\ r_{src}\ r_{dst} \mid \\
& op_{\oplus}\ r_1\ r_2\ r_{dst} \mid \textbf{load}\ r_{addr}\ r_{dst} \mid \textbf{store}\ r_{addr}\ r_{val} \mid \\
& \textbf{jump}\ r \mid \textbf{bnz}\ r\ i \mid \textbf{jal}\ r
\end{aligned}
$$

The semantics of instructions are standard (full details are in the extended version.): **nop** performs no operation, **halt** halts execution, and **out** r outputs the content of register r. Instructions of the form **const** i r load constant $i \in \mathbb{W}$ into register r, $op_{\oplus}$ r_1 r_2 r_{dst} combines registers r_1 and r_2 using operator $\oplus$ and stores the result in register r_{dst}. Memory access is performed using **load** and **store** instructions where **load** r_{addr} r_{dst} loads from the address in r_{addr} into r_{dst} and **store** r_{addr} r_{val} writes the value in r_{val} to the memory address stored in r_{addr}. The control flow can be manipulated using **jump** r, which jumps to the address in register r, or **bnz** r i, which jumps to address i if the value in r is non-zero. Function calls can be performed using an instruction of the form **jal** r which stores the current program counter at the address pointed to by a special register sp and jumps to the address in register r.

A configuration (mem, reg, pc) of this machine consists of a memory state $mem : \mathbb{W} \to \mathbb{W}$, a register state $reg : Reg \to \mathbb{W}$, and a program counter $pc \in \mathbb{W}$. Evaluation of configuration (mem, reg, pc) in one step to configuration (mem', reg', pc') while producing trace $\alpha \in \mathbb{W}^*$ is denoted by $(mem, reg, pc) \xrightarrow{\alpha} (mem', reg', pc')$. Machine code instructions are also encoded as machine words; $decode(w)$ turns a machine word into a symbolic representation of the instruction (if possible), while $encode(i)$ maps instruction i to the corresponding machine word. A program (is, pc_0) for this machine consists of a list of machine words $is \in \mathbb{W}^*$ together with a word indicating the expected starting address. A starting machine configuration for (is, pc_0) is produced from an initial memory mem and initial register state reg by overriding the words at $pc_0, \ldots, pc_0 + |is| - 1$ by the instructions is and setting the program counter to pc_0. We denote this by $(mem[pc_0 \mapsto is], reg, pc_0)$.

We assume that each memory location $a \in \mathbb{W}$ has an associated security level $\Gamma(a) \in \mathcal{L}$; similarly, each register $r \in Reg$ is associated with a security level $\Gamma(r) \in \mathcal{L}$.

We instantiate the parameters from Section 2.2 as follows:

i. States consist of a function mapping addresses, represented as machine words, to words and a function mapping registers to words: $State = (\mathbb{W} \to \mathbb{W}) \times (Reg \to \mathbb{W})$. The commands are given by a sequence of instructions together with a starting address $Com = \mathbb{W}^* \times \mathbb{W}$. A configuration (mem, reg, pc) is a triple, consisting of a memory state mem, a register state reg, and a program counter pc. I.e., $Conf = (\mathbb{W} \to \mathbb{W}) \times (Reg \to \mathbb{W}) \times \mathbb{W}$.

ii. The evaluation relation $\to$ is standard (details in the extended version).

iii. $(mem_1, reg_1) =_{\mathbf{L}} (mem_2, reg_2)$ holds iff $\forall a.\ \Gamma(a) = \mathbf{L} \Rightarrow mem_1(a) = mem_2(a)$ and $\forall r.\ \Gamma(r) = \mathbf{L} \Rightarrow reg_1(r) = reg_2(r)$.

iv. Extracting the state of a configuration returns the memory and registers: $state((mem, reg, pc)) = (mem, reg)$. Extracting the next instruction returns the current instruction and the value of the program counter: $com((mem, reg, pc)) = (mem[pc], pc)$. Note that this does not extract the entire program; only the next instruction to be executed is returned.
Constructing a new configuration is defined by

$$\langle (is, pc_0), (mem, reg) \rangle = (mem[pc_0 \mapsto is], reg, pc_0)$$

where $mem[pc_0 \mapsto is]$ denotes replacing the words at $pc_0, pc_0 + 1, \ldots, pc_0 + |is|$ by $is_0, is_1, \ldots, is_{|is|}$.

The extracted functions model the effect the various instructions have on the memory and registers. For example, the instruction **mov** r_s r_d results in $f((mem, reg)) = ((mem, reg[r_d \mapsto reg[r_s]]), \varepsilon)$, while **store** r_a r_v generates the state transformer $f((mem, reg)) = ((mem[reg[r_a] \mapsto reg[r_v]], reg), \varepsilon)$. Performing an output instruction **out** r induces the function $f((mem, reg)) = ((mem, reg), reg[r])$.

Note that mixing instructions and memory to create a configuration rules out some invalid initial states: For example, consider the following program that outputs the value of $(mem[5] - x) \times mem[h]$, for some constant x.

```
1 const 5 r1          // load constant 5 into r1
2 load r1 r2          // load mem[5] into r2
3 const x r1          // load constant x into r1
4 sub r2 r1 r2        // r2 = mem[5] - x
5 const h r1          // load constant h into r1
6 load r1 r3          // load mem[h] into r3
7 mul r2 r3 r2        // r2 = (mem[5] - x) * mem[h]
8 out r2              // output (mem[5] - x) * mem[h]
```

In the case where $x = encode(\textbf{load}\ r_1\ r_3)$, the program always produces the trace $[0]$, since this instruction will always be inserted at address 6 when creating a starting configuration. The function that is extracted from a run of this program is

$$f((mem, reg)) = ((mem, reg), [(mem[6] - x) \times mem[h]]$$

which will produce a trace different from $[0]$ if $mem[6] \neq encode(\textbf{load}\ r_1\ r_3)$. However, such a state does not correspond to a valid execution of the program and should not be considered when judging its security. This is addressed by the notion of *valid initial states* from Section 2.2. For any machine-code program $(is_0 \ldots is_n, pc_0)$, it holds that $\mathcal{S}_0((is, pc_0)) = \{(mem, reg) \mid mem[pc_0] = is_0, \ldots mem[pc_0 + |is| - 1] = is_n\}$, thereby ruling out impossible starting states when determining whether or not a program satisfies explicit secrecy.

Recall the key intricacy connected to applying weak secrecy to low-level programs: programs exist in the memory and their instructions can then be read and used for computations. As a simplified example, on how this can affect a security analysis, consider the following program:

```
1   const 3 r1        // put 3 into r1
2   load r1 r2        // load from mem[r1] into r2
3   bnz 17 r2         // assume a non-zero opcode
4   const r1 x        // load constant x into r1
5   sub r2 r1 r2      // set r2 := mem[3] - x
6   const h r1        // load address h into r1
7   load r4 r1        // load mem[r4] into r1
8   mul r1 r4 r2      // set r2 := (mem[3] - x) * mem[h]
9   out r2            // output (mem[3] - x) * mem[h]
```

This program reads the opcode of instruction 3, i.e., loads the value $encode(\textbf{bnz}\ 17\ r_2)$ into register r_2, subtracts

a constant x and outputs the result of multiplying r_2 with a value from a high memory location h. If we set $x = encode(\textbf{bnz}\ 17\ r_2)$, then every run of this program will produce [0]. An adaptation of weak secrecy to the machine code would involve modifying the executed sequence of instructions by eliminating branching instructions such as **bnz** 17 r_2. But such a modification might well result in a different value than $encode(\textbf{bnz}\ 17\ r_2)$ being loaded into register r_1 in the second instruction, resulting in a trace that depends on the value of h and renders the program insecure. In contrast, explicit secrecy handles such programs seamlessly since the same memory as in the original execution is used.

2.5. Explicit Secrecy in the Big Picture

We close the discussion of soundness conditions for taint tracking by examining the relation between explicit secrecy and standard notions from the area of information-flow control. We present an intuitive property aimed at capturing explicit flows as a variant of *gradual release*, a knowledge-based approach to noninterference with a declassification policy, and we review how explicit secrecy relates to noninterference. We discuss this approach in the context of the while language from Section 2.1, but it should generalize straightforwardly to other settings, such as machine code.

Intuitively, allowing information leakage due to implicit flows can be thought of as releasing (or *declassifying*) information about control-flow decisions taken during the run of a program; i.e., any high information used to determine control flow is considered to be disclosed to the attacker, but they should learn nothing else. To accommodate this form of declassification, we extend traces with *release events* of the form $rel(v)$ where $v \in \mathbb{Z}$. Every control-flow decision then generates an event of this form to release information about implicit flows to the attacker. Concretely, we modify the rules for control-flow commands in the semantics to produce release events. Below we show the rules for the "true" case for **if** and **while** commands (the other cases produce similar events).

$$\frac{m(e = 0) = n \qquad n = 0}{(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, m) \xrightarrow{rel(n)} (c_1, m)} \quad \text{(E-IfTrue)}$$

$$\frac{m(e = 0) = n \qquad n = 0}{(\textbf{while}\ e\ \textbf{do}\ c, m) \xrightarrow{rel(n)} (c; \textbf{while}\ e\ \textbf{do}\ c, m)} \quad \text{(E-WhileTrue)}$$

The intuition is then to allow the attacker's knowledge to increase based on release events, but not at any other point during the execution. We first define the knowledge of the attacker in the standard way [12]:

2.5.1 Definition [Knowledge]: The *knowledge set* for statement c, initial memory m_0, and trace ℓ is given by $k(c, m_0, \ell) = \{m \mid m =_{\mathbf{L}} m_0 \wedge (\exists c', m'.\ (c, m) \xrightarrow{\ell}{}^{*} (c', m'))\}$.

Based on this, we introduce *control-flow gradual release*, specifying that changes in knowledge are only allowed due to release events:

2.5.2 Definition [Control-flow gradual release]: A program c satisfies *control-flow gradual release*, written $CFGR \vDash c$, iff whenever $(c, m) \xrightarrow{\ell.\alpha}{}^{*} (c', m')$ and $\forall n.\ \alpha \neq r(n)$, we have $k(c, m, \ell) = k(c, m, \ell.\alpha)$.

As an example, consider again the program c_{impl} from Section 2.1, executed with some initial memory m_0 where $m_0(h) = 0$. Initially, $k(c_{impl}, m_0, \varepsilon) = [m_0]_{\mathbf{L}}$. After performing the branching for **if** $(h = 0)$, the release event $r(1)$ is produced, changing the knowledge set to $k(c_{impl}, m_0, [r(1)]) = \{m | m =_{\mathbf{L}} m_0 \wedge m(h) = 0\}$. Performing the output **out** 1 produces the event 1. However, the knowledge set remains unchanged: $k(c_{impl}, m_0, [r(1), 1]) = k(c_{impl}, m_0, [r(1)])$. The knowledge changes in an analogous way if $m_0(h) \neq 0$.

Control-flow gradual release is more permissive than explicit secrecy:

2.5.3 Theorem: If $ES \vDash c$, then $CFGR \vDash c$.

Perhaps surprisingly, this inclusion is strict: there are programs satisfying control-flow gradual release but not explicit secrecy. For example, consider a program that leaks the same information twice, first using an implicit and then an explicit flow:

```
if (h = 0)
   then out 1
   else out 2;
out (h = 0)
```

Upon reaching **out** $(h = 0)$, the knowledge set is already reduced to memories m where $m(h = 0) = m_0(h = 0)$. Thus, seeing the last output does not affect the knowledge of the attacker.

This program, however, is not accepted by explicit secrecy, as the extracted functions have the form $f(m) = (m, [i, m(h = 0)])$ for either $i = 1$ or $i = 2$, which produce different traces depending on whether or not $h = 0$. Practical taint-tracking systems would also reject this program.

What about the relation to ordinary noninterference? As Volpano pointed out, weak secrecy (and therefore also explicit secrecy), despite its name, is in fact *not* weaker than noninterference. For example, consider the following program (c_{ni}):

```
if (h = 0)
   then out h
   else out 0
```

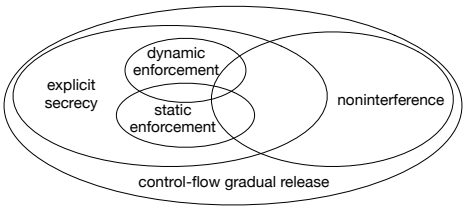
In both branches, this program will produce the trace [0]. However, one sequence of outputs and assignments that can be extracted is **out** h and hence $WS \nvDash c_{ni}$.

In the case of straight-line programs, noninterference and weak secrecy do coincide. In particular this entails that weak secrecy and explicit secrecy, like noninterference, cannot be expressed as a property on traces [44]: both conditions are hyperproperties [25].

Since control-flow gradual release relaxes noninterference using a fixed declassification policy, the former is weaker than the latter:

2.5.4 Theorem: If $NI \vDash c$, then $CFGR \vDash c$.

The relations between the various properties can be summarized as follows. (We will see the enforcement mechanisms in Section 3.)



3. Enforcement

A large body of research literature shows that static and dynamic taint tracking can mitigate a wide range of confidentiality and integrity vulnerabilities. Applications of taint analysis cover the full range of the hardware and software stack, including memory safety violations such as buffer overruns in low-level code, injection attacks such as cross-site scripting in web applications, and recently also privacy leaks in smart phone apps. On the practical side, numerous existing tools and languages substantiate the applicability of taint tracking in these settings. For instance, binary analysis platforms such as BAP [19] and BitBlaze [56] implement taint tracking for analyzing native applications. Languages such as Perl [4] and Ruby [3] have built-in support for taint analysis, while language extensions such as Andromeda [60] and Pixy [39] support tainting in web applications. In the mobile context, TaintDroid [34] and FlowDroid [8] leverage taint tracking to enforce security for phone apps.

Despite the widespread usage of taint tracking, there has been little effort to formally define policies and mechanisms implemented by these tools. Notably, Schwartz et al. [54] and Livshits [42] formalize the essence of dynamic taint tracking by instrumenting the operational semantics of a core language for assembly and Java programs, respectively. Recent work by Bodin et al. [18] presents a technique for deriving semantic dependency analyses from a natural semantics of an imperative language with objects, which stands at the core of taint analysis. While this line of works provides useful insight for implementing correct taint analysis, it does not formally justify soundness for a security condition like weak secrecy. Such a condition however can help to understand what policies taint tracking ensures.

Our goal in this section is to check that weak secrecy and explicit secrecy correctly capture the intuition of existing taint-tracking systems. We prove soundness of flow-sensitive dynamic taint tracking for high-level code (Section 3.1) and machine code (Section 3.2) for explicit secrecy. Further, we present a static analysis for Java-like code based on symbolic execution and automated theorem proving and prove it sound for explicit secrecy (Section 3.3).

3.1. Dynamic Tainting for Imperative Code

We first discuss taint tracking in the context of the while language introduced in Section 2.1 and prove it sound with respect to the instantiation of explicit secrecy as described in Section 2.4.1.

A flow-sensitive dynamic taint tracker, as in real-world languages such as Perl or Ruby, keeps track of what variables have tainted content at each point during execution. To formalize this, we extend configurations with a function $\tau : Var \to \mathcal{L}$ mapping program variables to security levels. We replace the evaluation rules for assignments and output as follows (the other rules simply propagate the taints unchanged, as illustrated by T-IfTrue). As before, initial security labeling of program inputs, i.e., memories, is determined by Γ and we write τ_0 for Γ.

$$\frac{m(e) = n}{(x := e, m, \tau) \hookrightarrow (\varepsilon, m[x \mapsto n], \tau[x \mapsto \tau(e)])} \text{ (T-Assign)}$$

$$\frac{m(e) = n \qquad \tau(e) = \mathbf{L}}{(\textbf{out } e, m, \tau) \overset{n}{\hookrightarrow} (\varepsilon, m, \tau)} \text{ (T-OutL)}$$

$$\frac{\tau(e) = \mathbf{H}}{(\textbf{out } e, m, \tau) \hookrightarrow \lightning} \text{ (T-OutFail)}$$

$$\frac{m(e = 0) = n \qquad n = 1}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, m, \tau) \hookrightarrow (c_1, m, \tau)} \text{ (T-IfTrue)}$$

Figure 1. Dynamic Tainting for Imperative Code

When evaluating an assignment $x := e$, the taint state $\tau(x)$ of variable x is updated to match the taint of the expression that is being assigned to x. We extend τ to arbitrary expressions by defining $\tau(e) = \bigsqcup_{x \in vars(e)} \tau(x)$ where $vars(e)$ denotes the set of variables appearing (syntactically) in e. If a tainted variable occurs in an expression that is output, then the execution is stopped (written $\lightning$). (Some tainting tools [47] use more precise analyses that can avoid fake dependencies such as $l := h - h$.) We can now show that whenever the dynamic taint tracker reaches a configuration (c', m') starting from an initial configuration (c, m), the program c satisfies explicit secrecy with respect to that run.

3.1.1 Theorem [Soundness of Tainting]: For any program c, initial state m, and initial security labeling τ_0, if $(c, m, \tau_0) \not\hookrightarrow^* \lightning$, then $ES \vDash (c, m)$.

3.2. Dynamic Tainting for Machine Code

We now show how to enforce explicit secrecy by a flow-sensitive dynamic taint tracker for the machine language presented in Section 2.4.2. Analogously to taint tracking for imperative programs, we extend machine configurations with another component keeping track of which memory addresses and registers contain tainted data. The taint state is represented by a function $\tau : Reg \cup \mathbb{W} \to \mathcal{L}$ which maps registers and memory addresses to security levels. This function records the taint status for all registers and memory addresses, and uses them to derive the taint status for all

$$\frac{decode(mem[pc]) = \textbf{out}\ r \qquad \tau(r) = \mathbf{H}}{(mem, reg, pc, \tau) \xrightarrow{reg[r]} \lightning} \text{(T-OUTFAIL)}$$

$$\frac{decode(mem[pc]) = \textbf{out}\ r \qquad \tau(r) = \mathbf{L}}{(mem, reg, pc, \tau) \xrightarrow{reg[r]} (mem, reg, pc + 1, \tau)} \text{(T-OUTL)}$$

$$\frac{\begin{array}{c} decode(mem[pc]) = \textbf{jal}\ r \\ a = reg[sp] \qquad mem' = mem[a \to pc] \end{array}}{\begin{array}{c} (mem, reg, pc, \tau) \hookrightarrow \\ (mem', reg, reg[r], \tau[a \mapsto \tau(sp) \sqcup \tau(pc)]) \end{array}} \text{(T-JAL)}$$

$$\frac{\begin{array}{c} decode(mem[pc]) = \textbf{load}\ r_a\ r_d \\ v = mem[reg[r_a]] \end{array}}{\begin{array}{c} (mem, reg, pc, \tau) \hookrightarrow \\ (mem, reg[r_d \mapsto v], pc + 1, \tau[r_d \mapsto \tau(r_a) \sqcup \tau(reg[r_a])]) \end{array}} \text{(T-LOAD)}$$

Figure 2. Dynamic Tainting for Machine Code

values during the execution. Figure 2 presents an excerpt of the instrumented semantics rules for a dynamic taint tracker that handles self-modifying machine code (the extended version gives the full set of rules). Our formalization resembles the ones used by real-world dynamic taint trackers such as those implemented by the CMU Binary Analysis Platform (BAP) [19] and the BitBlaze binary analysis platform [56]; it can also cope with self-modifying code (which they cannot). Schwartz et al. [54] formalize dynamic taint analysis by means of operational semantics for a low level language used in BAP and BitBlaze. Most of the challenges involve soundness of the taint-tracking mechanisms, especially undertainting. Explicit secrecy allows deciding which design choices result in unsoundness, making it easier to show whether this is appropriate in a particular scenario. E.g., time-of-detection vs. time-of-attack issues can be addressed by appropriately choosing the attacker observations.

The taint state τ is updated as for the imperative programs. If a tainted register is used to output a value (rule T-OUTFAIL), the execution is stopped. Otherwise the program performs the output and the execution proceeds (rule T-OUTL). Rule T-JAL updates the taint state by applying the join operator to combine the taint status of the address pointed by register sp with the taint status of the current program counter pc. Rule T-LOAD combines the taint status of address it loads from with the taint status of target register and assigns the resulting label to that register.

The following theorem shows that whenever the instrumented semantics in Figure 2 is accepted by taint tracking (i.e., it does not fail because of a violation of the taint policy), the original program satisfies explicit secrecy.

3.2.1 Theorem [Soundness of Tainting]: For any program $c = (is, pc_0)$, initial state $s_0 = (mem_0, reg_0)$ and initial security labeling τ_0, if $(mem_0[pc_0 \mapsto is], reg_0, pc_0, \tau) \not\hookrightarrow^* \lightning$ then $ES \vDash (c, s_0)$.

3.3. Static Analysis for Taint Tracking

Lastly, we present a static analysis for enforcing explicit secrecy for imperative programs with a heap. The analysis leverages symbolic execution and theorem proving to statically certify the security of simple Java-like programs with respect to explicit secrecy. Static analysis stands at the core of several taint tracking tools. For instance, Andromeda and FlowDroid perform static taint analysis for web and mobile applications. While Andromeda's and FlowDroid's static analysis target the more challenging setting of real Java code, we extend our simple imperative language from Section 2.1 with a heap to provide a static analysis that checks for explicit secrecy. This new language is the same as the one used to describe the essence of static taint analysis implemented by the tools above [60], [8].

The language from Section 2.1 is extended with heap locations and fields, object creation, and load and store expressions on object fields.

$$c ::= \quad \cdots \mid x := \textbf{new}\ Object() \mid x := y.f \mid x.f := y$$

We extend the set of values Val with a set of object locations Loc and a **null** value. A configuration (c, m, h) consists of a program c, a memory $m \in Mem$ and a heap $h \in Heap$ mapping locations and field identifiers to values, i.e. $Heap = Loc \times Fld \to Val$. The evaluation relation is extended as expected; details can be found in the extended version.

We reason about the behavior of a Java-like program by means of forward symbolic execution. Symbolic execution allows us to build a logical formula, which corresponds to a set of concrete program executions, and use first-order reasoning to statically prove whether or not that program satisfies explicit secrecy. We assume that there is a sound (but not necessarily complete) procedure for determining validity of first-order formulas. Concretely, such a procedure could be implemented by an SMT solver. We denote that a formula φ is reported as valid by SMT $\vdash \varphi$. The program is executed on symbolic inputs, hence the state and the configuration are also symbolic.

A symbolic configuration $\langle c, \delta, \varphi \rangle$ consists of a command c, a symbolic state δ, and a path condition φ, where φ is a (quantifier-free) first-order formula. A symbolic state $\delta : (Var \to Expr) \cup (Loc \times Fld \to Expr)$ is a mapping from program variables and object fields to expressions. The intuition is that $\delta(x)$ expresses the value of a variable x or an object field $y.f$ at some point of the execution in terms of the values of initial variables. The symbolic state is updated when processing assignments to reflect changes to variables or fields. For example, after performing the assignments $x := y\ ;\ x := x + z$, the symbolic state records the value of x as $\delta(x) = y + z$. Similarly, the program $x := \textbf{new}\ Object()\ ;\ x.f := y$ yields a symbolic state δ such that $\delta(x) = l$, for some fresh location $l \in Loc$, and $\delta(l, f) = y$. A path condition φ is a symbolic boolean expression built over the initial variables and it constrains the set of concrete initial states to those that execute a given program path.

Figure 3 presents an excerpt of the symbolic evaluation rules for statically checking explicit secrecy for the Java-like language. We denote a symbolic expression e where all high (tainted) variables have been renamed (deterministically) to fresh variables by $\tilde{e}$. Consider for example a boolean expression $e = (l + h > 0)$ such that $\Gamma(l) = \mathbf{L}$ and $\Gamma(h) = \mathbf{H}$. Then $\tilde{e} = (l{+}h')$ for some fresh variable h'. This simulates a second run with a low-equivalent memory in the style of self-composition [15]. Output instructions **out** e are validated by checking for equivalence of the expression in the two memories by requiring that $\delta(e) = \widetilde{\delta(e)}$. The merging of two symbolic states δ_1 and δ_2 when branching on expression e is defined by $(\delta_1 \oplus_e \delta_2)(x) = \delta_1(x)$ if $\delta_1(x) = \delta_2(x)$ and $(\delta_1 \oplus_e \delta_2)(x) = (e \ ? \ \delta_1(x) : \delta_2(x))$ if $\delta_1(x) \neq \delta_2(x)$. The merging operation allows for a compact representation of symbolic state modifications occurring in each branch: Instead of tracking modifications for each branch separately, we encode changes that occurred in each branch as one expression. For example, after the command **if** e **then** $x := 1$ **else** $x := 2$, the variable contains either 1 or 2 depending on the branch that was taken. To avoid exploring the rest of the program for both possible values of x, we encode this dependency as $\delta(x) = (e \ ? \ 1 : 2)$. At the same time, the branch condition is only needed to check whether branches are reachable. This analysis benefits from recording precise dependency information about x instead of just tracking just 1 and 2 as possible values.

In order to simulate executions taking the same branches, control-flow information has to be removed from δ. We introduce a function $\mathit{forget}(\cdot) : \mathit{Expr} \rightarrow \mathit{Expr}$, where $\mathit{forget}(e)$ removes information resulting from merging of symbolic states after **if** statements. For example, consider the implicit flow example, c_{impl}. The dependency recorded for l after performing the assignment in the branch is $\delta(l) = (h \ ? \ 1 : 2)$. Since this expression depends on h, checking $(h \ ? \ 1 : 2)$ and $(h' \ ? \ 1 : 2)$ for equivalence will fail, leading to the program being rejected the program as insecure. The reason for this is that control-flow information about how l was modified is present in δ. Concretely, the solver would fail to show validity of the formula $((\textbf{if } h \textbf{ then } 1 \textbf{ else } 2) = (\textbf{if } h' \textbf{ then } 1 \textbf{ else } 2))$, which does not hold, for example, when $h = \neg h'$. Such information is removed by recursively replacing all occurrences of $(e' \ ? \ e_1 : e_2)$ in e by $(v \ ? \ e_1 : e_2)$, where v is a fresh low variable. A low variable is introduced to force the program to take the same branch in both executions; the variable is fresh so that both branches will be considered by the prover. In fact, the formula $((\textbf{if } v \textbf{ then } 1 \textbf{ else } 2) = (\textbf{if } v \textbf{ then } 1 \textbf{ else } 2))$ is now valid.

The rules in Figure 3 force the symbolic execution engine to apply a breadth-first search strategy for analyzing the program. Rule S-OUT considers an output expression e as secure whenever the state transformation leading to that expression is unaffected by initial tainted values. Indeed, the validity of the first-order formula $\mathit{forget}(\delta(e)) = \mathit{forget}(\widetilde{\delta(e)})$ forces all initial concrete memories that start with the same values for untainted variables (as enforced by the renaming in $\widetilde{\delta(e)}$) and follow the same execution path (as enforced by function $\mathit{forget}(\cdot)$) to always output the same value to the attacker. This implies that the output observed by the attacker is unaffected by initial tainted values, as required by explicit secrecy. Rule S-IF symbolically evaluates each branch of a conditional and merges the respective symbolic states if the execution of both branches succeeds. Rule S-WHILEUNROLL unrolls a **while** loop one time. (Unless the number of loop iterations is known a priori, this rule may apply indefinitely, leading to nontermination. Some tools apply loop unrolling for a fixed number of iterations and leave open the possibility of false negatives. This approach is usually taken whenever a tool is used for bug finding, where full automation is more important than possible unsoundness.) Rule S-NEW creates a fresh object location that maps the program variable to that location. Rules S-LOAD loads the symbolic expression contained at some object field by first looking up the object location and then the field value of that object location, while rule S-STORE stores the symbolic expression of some program variable to an object field.

$$\frac{\text{SMT} \vdash \mathit{forget}(\delta(e)) = \mathit{forget}(\widetilde{\delta(e)})}{\langle \textbf{out } e, \delta, \varphi\rangle \rightsquigarrow (\varepsilon, \delta, \varphi)} \quad \text{(S-OUT)}$$

$$\frac{}{\langle x := e, \delta, \varphi\rangle \rightsquigarrow \langle \varepsilon, \delta[x \mapsto \delta(e)], \varphi\rangle} \quad \text{(S-ASSIGN)}$$

$$\frac{\begin{array}{c}\langle c_1, \delta, \varphi \wedge \delta(e)\rangle \rightsquigarrow^* \langle \varepsilon, \delta_1, \varphi_1\rangle \\ \langle c_2, \delta, \varphi \wedge \neg\delta(e)\rangle \rightsquigarrow^* \langle \varepsilon, \delta_2, \varphi_2\rangle \\ \text{SMT} \not\vdash \varphi \rightarrow \delta(e) \qquad \text{SMT} \not\vdash \varphi \rightarrow \neg\delta(e)\end{array}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \delta, \varphi\rangle \rightsquigarrow \langle \varepsilon, \delta_1 \oplus_{\delta(e)} \delta_2, \varphi\rangle} \quad \text{(S-IF)}$$

$$\frac{}{\begin{array}{c}\langle \textbf{while } e \textbf{ do } c, \delta, \varphi\rangle \rightsquigarrow \\ \langle \textbf{if } e \textbf{ then } (c \,;\, \textbf{while } e \textbf{ do } c) \textbf{ else } \varepsilon, \delta, \varphi\rangle\end{array}} \quad \text{(S-WHILEUNROLL)}$$

$$\frac{l \in \mathit{Loc} \text{ fresh} \qquad \delta' = \delta[x \mapsto l]}{\langle x := \textbf{new } \mathit{Object}(), \delta, \varphi\rangle \rightsquigarrow \langle \varepsilon, \delta', \varphi\rangle} \quad \text{(S-NEW)}$$

$$\frac{l = \delta(y) \qquad \delta' = \delta[x \mapsto \delta(l, f)]}{\langle x := y.f, \delta, \varphi\rangle \rightsquigarrow \langle \varepsilon, \delta', \varphi\rangle} \quad \text{(S-LOAD)}$$

$$\frac{l = \delta(x) \qquad \delta' = \delta[(l, f) \mapsto y]}{\langle x.f := y, \delta, \varphi\rangle \rightsquigarrow \langle \varepsilon, \delta', \varphi\rangle} \quad \text{(S-STORE)}$$

Figure 3. Static Analysis for Taint Tracking

Static taint trackers which are more tailored towards verification would attempt either to infer a loop invariant or ask the user to provide one manually. The inference of loop invariants is discussed in the extended version of the paper. We then show soundness of the static enforcement: whenever the symbolic execution engine terminates successfully, the program satisfies explicit secrecy.

3.3.1 Definition: A program c satisfies *static enforcement*, written $\vdash_s c$, iff there exist δ', φ' such that $\langle c, \delta_0, \top\rangle \rightsquigarrow^* \langle \varepsilon, \delta', \varphi'\rangle$.

3.3.2 Theorem [Soundness of Static Analysis]: If $\vdash_s \ c$ then $ES \vDash c$.

Note that this static enforcement technique can establish security of some programs that are rejected by the dynamic monitoring presented in Section 3.1, such as **out** $(h - h)$. Depending on the power of the SMT solver, static enforcement can fail to validate a program due to dead code. For example, consider **if** e **then out** h **else out** 0, where e is a complex expression that always returns 0. If the solver fails to establish that e is equivalent to 0, the program will be rejected. SMT solvers can indeed be expensive, especially for languages pointers and higher-order features. Perhaps this can be remedied by leveraging sound data-flow analysis (à la FlowDroid) or security type systems (as in Section 3.2). We have preliminary experiments along these lines with simple programs (no quantifier alternation or non-linear arithmetic); however, over-approximation is needed for richer languages.

Compared to the tainting algorithms used to justify soundness of Andromeda and FlowDroid, the analysis proposed in this section is more precise, i.e., it accepts more secure programs. The rationale behind this choice is that for complex languages as Java, scalability may become an issue as precision increases. Indeed, one central problem in the static analysis of production code is to reconcile precision (object-, field-, context-, flow-sensitivity), scalability (lines of code analyzed) and soundness (FlowDroid is unsound for certain language features). FlowDroid achieves this goal using the IFDS framework by Reps et al. [49] to implement a modular inter-procedural data-flow analysis with on-demand aliasing.

4. Related Work

Taint-tracking Policies. Our framework is the first to offer a general characterization of policies for taint tracking. The direct precursor of our approach, weak secrecy [63], relies on extracting assignment commands from the original program and, due to its syntactic nature, requires custom-tailored extensions to address such language features as reflection and expressions with side effects.

Chaudhuri et al. [21] implement a calculus for data-flow integrity on Windows Vista. They use a type system and runtime checks to conservatively enforce a tainting policy with respect to an instrumented operational semantics. The instrumented semantics enriches standard semantics with security labels allowing to express explicit secrecy as a safety property. As discussed earlier, explicit secrecy is a hyperproperty [25], hence our condition can be used to justify the soundness of their enforcement mechanism with respect to uninstrumented semantics.

Livshits and Chong [43] address the problem of automatic placement of sanitizers through hybrid taint analyses. They express security policies in terms of sanitizers for a source-sink pair and apply taint tracking to an interprocedural data-flow graph to place the sanitizers. The correctness criteria for tainting algorithms state that sanitizers are placed as described by the policy. Livshits [42] provides a taxonomy of dynamic taint tracking approaches for high-level languages, mainly targeting web security. Taint tracking is an instance of data-flow analysis [40], a method for computing properties about data by observing how it flows through the program. Static and dynamic data-flow analysis, ranging from security types to symbolic executions, have been applied to enforcing security [54], [42], [18]. The semantic notion of explicit secrecy can serve as a criteria for validating the correctness of these approaches.

Information-flow Policies. A large array of works on security policies address information-flow control. These policies include the baseline notion of noninterference [35], and variations which account for different policies, languages and computation models [51]. The policies proposed in this paper stand to taint-tracking mechanisms as noninterference-like policies stand to information-flow control mechanisms.

As discussed earlier, there has been work on exploring connections between information-flow control and taint tracking. Denning and Denning [32] are the first to distinguish between explicit (as in tainting) and implicit flows. Russo et al. [50] show how a lightweight control-flow graph analysis can be combined with explicit flow analysis for non-malicious code. Coppens et al. [29] leverage explicit flow analysis for information-flow security by selective if-conversion to remove branching on secrets in programs by transformation [29]. This technique is a particularly good fit for the implementation of cryptographic algorithms where transforming away branching helps mitigating the timing side channel [14].

A line of work by Vachharajani et al. [61], Graa et al. [36], Beringer [16], and Shen et al. [55] utilizes taint tracking for information-flow control by turning control-flow checks for a source program into data-flow checks of the resulting program. The control-flow checks can be injected by program transformation so that taint tracking on the resulting program can be used for tracking information flow in the source program.

Our knowledge-based condition draws on the notion of *gradual release* introduced by Askarov and Sabelfeld [12]. Knowledge-based conditions have also been used to provide intuitive semantics for dynamic information-flow policies [9], [62]. This is done by considering attackers that partially forget the observations made during the computation. Although the tainting attacker is forgetful in the sense that it only recalls a single observable event, a precise characterization of tainting policies in terms of forgetful attacker knowledge is non-obvious because it requires to encode attackers that forget the control flow of the program.

Dynamic/Hybrid Enforcement. Taint analysis provides a good balance between implementation effort, bug coverage and performance overhead. This has led to a pervasive application of dynamic taint tracking for enforcing security at all levels of hardware and software stacks. Many systems implement tainting mechanisms in hardware by adding architectural extensions to processors [58], [30], [23], [33] or in software by extending and instrumenting machine code [47], [22], [56], [26], web and mobile applications [34], [46], [59], [43], [31] or high-level languages [4], [3].

Customized hardware and emulators have been used to implement taint tracking policies. Suh et al. [58] introduce a hardware mechanism for dynamically tracking information flows in a program. On every instruction, the processor determines whether the result is tainted or not based on the inputs and the instruction type. Various vulnerabilities such as buffer overflows and format strings are captured by disallowing tainted data to be used as instructions or jump target addresses. Crandall et al. [30] introduce Minos, a microarchitecture that implements Biba's integrity policies at word level. Minos tracks the integrity of all data and it protects from control flow hijacking by checking taintedness whenever a program uses that data to transfer control. TaintBoch [23] uses tainting to track sensitive data across operating system, language, and application boundaries, thus permitting analysis at a whole system level. In principle, all these approaches can handle self-modifying code and thus benefit from our security condition to justify their correctness.

Many tools use dynamic instrumentation of machine code to monitor system activities through taint tracking. TaintTrace [22] uses the DynamoRIO framework [1] to instrument machine code. Similarly, TaintCheck [47] offers both Valgrind-based [5] and DynamoRIO-based implementations for the same purpose. Notably, the CMU Binary Analysis Platform [19] and the BitBlaze toolchain [56] offer a unified platform for static and dynamic security analysis of binary code. They explicitly represent all side effects of machine instructions in an intermediate language (IL), for which various forms of data-flow analysis, including taint tracking, are implemented. Assuming correctness of the transformation between binary code and IL code, one may use the well-defined semantics of IL to show soundness for the data-flow analysis implemented by these tools.

Dynamic taint tracking has been successfully used for bug finding in mobile and web applications. TaintDroid [34] is an extension to the Android platform that monitors how a potentially untrusted application uses user data. TaintDroid uses dynamic tainting to automatically track propagation of sensitive data through program variables or files and ensures that tainted data are transmitted over the network only with user's consent.

Perl's taint mode [4] is one of the first applications of dynamic taint tracking to high-level languages. When the interpreter is run in this mode, several data sources such as environment variables or command-line parameters produce tainted values. Tainted data may not be used by any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes, with a few exceptions: (1) arguments to `print` and `syswrite` are not checked for taintedness; (2) symbolic methods and symbolic sub references are not checked for taintedness; and (3) hash keys are never tainted. These exceptions allow for laundering (i.e., declassifying) tainted values either by using them as hash keys or by regexp-matching against them and using the sub-match strings $1, $2, etc.

Several generic frameworks offer customizable data-flow analysis and policies to account for the lack of generality and the high performance overhead in traditional taint-tracking systems. GIFT [41] is a compiler for programs written in C that takes programmer-specified rules for taint initialization, propagation and combination, and automatically instruments programs so as to execute these rules as part of the program execution. Chang et al. [20] build a compiler to transform untrusted programs into policy-enforcing programs. The compiler can be reconfigured to support new analyses and policies and it uses static analysis reduce the amount of data that must be dynamically tracked. Our work lays ground for turning informal soundness claims in this work into formal ones.

Static Enforcement. Purely static analysis has been proposed to tracking explicit flows in various application domains. Bodden et al. [8] present FlowDroid, a context-, flow-, field-, object-sensitive static taint analysis tool for Android applications. FlowDroid analysis is claimed to be conservative and sound with respect to the analysis it implements. However, unsoundness can arise in case the sequential consistency of thread execution is broken, or through native methods that are possibly modeled incorrectly.

Sridharan et al. [57] present F4F, a system for effective taint analysis of framework-based web applications. F4F initially analyses application code and configuration files to generate specifications of framework-related behaviors, and then uses a taint engine to perform more precise analysis of the framework-based applications.

Pixy [39] is a static taint analysis tool for detecting cross-site scripting and SQL injection vulnerabilities in PHP programs. It implements flow-sensitive, interprocedural and context-sensitive data-flow analysis to discover vulnerable program points.

The Parfait bug checker [53] builds on top of the LLVM compiler, performing a staged dependency analysis which takes into account both data and control dependencies. The tool is inherently tunable to different precision levels and it allows for demand driven analysis.

Balliu et al. [13] implement an automated tool for information-flow analysis of machine code. The tool transforms ARMv7 binaries into an architecture-independent format using the BAP platform [19] by means of a verified translator. This approach leverages symbolic execution and SMT solvers for machine-code verification, hence it is possible to extend the symbolic algorithm in Section 3.3 to that setting and thus verify explicit secrecy.

5. Conclusion

We have presented a generic semantic framework for reasoning about explicit flows as tracked by taint checking. The framework generalizes previous work by giving a condition that enables reasoning about what taint tracking guarantees and covers a wide range of settings, declassification/sanitization policies, and including low-level machines and languages with reflection. The framework has allowed us to formally compare security conditions, establishing a relation to such known characterizations as noninterference

and gradual information release. We have demonstrated the usefulness of the framework by instantiations to high- and low-level languages.

Explicit secrecy contributes to understanding of what is achieved by popular taint tracking tools, both dynamic and static. We have demonstrated that taint mechanisms at the core of dynamic tools such as BAP [19] and BitBlaze [56] and languages as Perl [4] and Ruby [3] are sound with respect to explicit secrecy. Similarly, we have showed the soundness of a mechanism reminiscent of FlowDroid [8] to guarantee the absence of explicit flows.

Our work opens up promising opportunities to formalizing informal soundness claims made in the taint tracking literature, following the initial steps in Section 3. An important track for future work is generalizing the soundness results to include declassification/sanitization, as defined in Section 2.3. Section 4 discusses several future tracks for applying our approach to the static and dynamic enforcement mechanisms from the literature. The expected benefits are greater confidence in these mechanisms and possibly discovering corner cases where soundness can be improved.

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

References

[1] Dynamic instrumentation tool platform. http://www.dynamorio.org/home.html.

[2] IDS03-J. Do not log unsanitized user input - CERT Oracle Coding Standard for Java - CERT Secure Coding Standards. https://www.securecoding.cert.org/confluence/display/java/IDS03-J.+Do+not+log+unsanitized+user+input. Accessed: 2015-8-5.

[3] Locking ruby in the safe. http://phrogz.net/programmingruby/taint.html.

[4] Perl security and taint mode. http://perldoc.perl.org/perlsec.html.

[5] Valgrind. http://valgrind.org/.

[6] Explicit secrecy: A policy for taint tracking. Full version. http://www.cse.chalmers.se/~schoepe/tainting.html.

[7] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, 1999.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[9] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *CSF*, 2012.

[10] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[11] A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *ESOP*, 2010.

[12] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *S&P*, 2007.

[13] M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *CCS*, 2014.

[14] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, 2014.

[15] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *MSCS*, 2011.

[16] Lennart Beringer. End-to-end multilevel hybrid information flow control. In Ranjit Jhala and Atsushi Igarashi, editors, *APLAS*, 2012.

[17] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying facets of information integrity. In *ICISS*, 2010.

[18] M. Bodin, T. Jensen, and A. Schmitt. Pretty-big-step-semantics-based certified abstract interpretation (preliminary version). In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, 2013.

[19] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *CAV*, 2011.

[20] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS*, 2008.

[21] A. Chaudhuri, P. Naldurg, and S. K. Rajamani. A type system for data-flow integrity on windows vista. *SIGPLAN Notices*, 2008.

[22] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC*, 2006.

[23] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004.

[24] D. Clark and S. Hunt. Non-interference for deterministic interactive programs. In *FAST*, 2008.

[25] M. R. Clarkson and F. B. Schneider. Hyperproperties. *JCS*, 2010.

[26] J. A. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, 2007.

[27] E. S. Cohen. Information transmission in sequential programs. In *FSC*. 1978.

[28] J. J. Conti and A. Russo. A taint mode for Python via a library. In *NordSec*, 2010.

[29] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P*, 2009.

[30] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO*, 2004.

[31] M. Dam, G. Le Guernic, and A. Lundblad. Treedroid: a tree automaton based approach to enforcing data processing policies. In *CCS*, 2012.

[32] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 1977.

[33] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon. PUMP – A programmable unit for metadata processing. In *HASP*, 2014.

[34] W. Enck, P. Gilbert, S. Han, V. Tendulkar, Byung-Gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 2014.

[35] J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P*, 1982.

[36] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli. Detecting control flow in smarphones: Combining static and dynamic analyses. In *CSS*, 2012.

[37] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *ACSAC*, 2005.

[38] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*, 2014.

[39] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *JCS*, 2010.

[40] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 1976.

[41] Lap-Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC*, 2006.

[42] B. Livshits. Dynamic taint tracking in managed runtimes. Technical Report MSR-TR-2012-114, Microsoft, November 2012.

[43] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL*, 2013.

[44] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *S&P*, 1994.

[45] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[46] Netscape. Using data tainting for security. http://www.aisystech.com/resources/advtopic.htm, 2006.

[47] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[48] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[49] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[50] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. *Marktoberdorf Summer School (IOS Press)*, 2009.

[51] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 2003.

[52] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *JCS*, 2009.

[53] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *SCAM, 2008*, 2008.

[54] E. J. Schwartz, T. Avgerinos, and D. Brumley. 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.

[55] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In *UbiComp*, 2015.

[56] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.

[57] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In *OOPSLA*, 2011.

[58] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[59] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI*, 2009.

[60] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, 2013.

[61] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *MICRO*, 2004.

[62] B. van Delft, S. Hunt, and D. Sands. Very static enforcement of dynamic policies. In *POST*, 2015.

[63] D. M. Volpano. Safety versus secrecy. In *SAS*, 1999.