Let's Face It: Faceted Values for Taint Tracking

Daniel Schoepe[1], Musard Balliu[1], Frank Piessens[2], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology, Sweden
[2] iMinds-DistriNet, KU Leuven, Belgium

Abstract. Taint tracking has been successfully deployed in a range of security applications to track data dependencies in hardware and machine-, binary-, and high-level code. Precision of taint tracking is key for its success in practice: being a vulnerability analysis, false positives must be low for the analysis to be practical. This paper presents an approach to taint tracking, which does not involve tracking taints throughout computation. Instead, we include shadow memories in the execution context, so that a single run of a program has the effect of computing on both tainted and untainted data. This mechanism is inspired by the technique of secure multi-execution, while in contrast to the latter it does not require running the entire program multiple times. We present a general framework and establish its soundness with respect to explicit secrecy, a policy for preventing insecure data leaks, and its precision showing that runs of secure programs are never modified. We show that the technique can be used for attack detection with no false positives. To evaluate the mechanism in practice, we implement DroidFace, a source-to-source transform for an intermediate Java-like language and benchmark its precision and performance with respect to representative static and dynamic taint trackers for Android. The results indicate that the performance penalty is tolerable while achieving both soundness and no false positives on the tested benchmarks.

1 Introduction

Taint tracking has been successfully deployed in a range of security applications to track data dependencies in hardware [14,35] and binary [13,34] code, as well as high-level code, with popular usage in mobile [22,19,1,40,21,10] and web [28,36,26] applications.

Background Taint tracking is about tracking direct data dependencies, or *explicit flows* [16], when data is passed directly from one data container to another. Taint tracking typically ignores *implicit flows* [16], when the information flows through the control structure of the program, as in, e.g., branching on a secret and assigning to different publicly observable variables in the branches.

What makes taint tracking a popular security mechanism? Missing out on implicit flows is clearly a disadvantage from the security point of view. This makes taint tracking a vulnerability finding mechanism rather than a mechanism that provides comprehensive security assurance. This brings us to an important observation: *precision* of taint tracking is key for its success in practice: being a vulnerability analysis, false positives must be low for the analysis to be practical.

This observation is echoed by the state of the art on taint tracking for Android applications (detailed in Section 5). Static taint trackers (such as FlowDroid [1], Amandroid [40], DroidSafe [21], and HornDroid [10]) and dynamic taint trackers (such as TaintDroid [19] and AppFence [22]) incorporate increasingly sophisticated features to catch data leaks while reducing the false positives.

Problem Motivated by the above, we seek to devise a general technique for tracking data leaks with high precision. Our goal is to formally establish the soundness and precision as well as demonstrate them in practice, with taint tracking in Android applications as a target for case studies.

The general setting is a program that operates in an environment with information *sources* and *sinks*. The goal of taint tracking is to prevent information from sensitive sources to directly affect the information sent to insensitive sinks. For confidentiality, this corresponds to not directly propagating information from secret sources to public sinks. This is often a desirable goal in the context of Android apps, as in e.g. allowing an app to access the file system to choose an image for a user profile but ensuring that no other files are leaked. In the examples throughout the paper, we will stick to confidentiality policies, noting that taint tracking has also been used successfully for integrity checks, e.g., [14,35,13,34].

Facelifted values This paper presents an approach to taint tracking, which, somewhat surprisingly, does not involve tracking taints throughout computation. Instead, we include shadow memories in the execution context, so that a single run of a program has the effect of computing on both sensitive and non-sensitive data. We refer to such values that carry both secret data as well as a public shadow value as *facelifted values*, in reference to the *faceted value* approach [2] by Austin and Flanagan.

Consider a simple example program:

$$h \leftarrow \mathbf{in}(\mathbf{H}); l := h; \mathbf{out}(\mathbf{L}, l) \tag{1}$$

Secret, or *high* (**H**), input is stored in a variable h and is explicitly leaked into the variable l, which in turn is output on a public, or *low* (**L**), channel. In essence, our approach has the effect of running the program:

$h \leftarrow \mathbf{in}(\mathbf{H})$; $h' := d$; // secret input and shadow input with default value d
$l := h$; $l' := h'$; // original assignment and shadow assignment
$\mathbf{out}(l', \mathbf{L})$ // public output from shadow memory

The shadow memory is represented by the shadow variables h' and l'. This represents the public view of the system. On a secret input, a default value d is stored in the shadow variable h'. On a public output, the value is retrieved from the shadow memory.

Soundness and precision This mechanism is inspired by the technique of *secure multi-execution* (SME) [11,17], where programs are executed as many times as there are security levels, with outputs at each level computed by the respective runs. SME addresses both explicit and implicit flows, enforcing the

policy of *noninterference* [15,20] that prescribes no leaks from sensitive inputs to insensitive outputs.

In contrast to SME, our mechanism does not re-run the entire program, focusing on secure-multi execution of the individual side-effectful commands that cause explicit flows. Moreover, it is independent of the choice of scheduling strategy for different runs. As such, this technique is similar to Austin and Flanagan's *faceted values* [2]. Re-purposing faceted values to track explicit flows results in a powerful mechanism for a policy that the original faceted values were not intended for: *explicit secrecy* [33], a policy that captures what it means to leak information explicitly. Further, facelifted values are different in that: i) Faceted values face challenges with tracking implicit flows, which results in propagating context labels through the computation. ii) Facelifted values are sound and precise for explicit secrecy, while faceted values are sound and *not* precise for noninterference [6]. iii) Facelifted values only require a single path through the program, while faceted values may execute both branches of a conditional [2]. iv) As a consequence, facelifted values can be implemented by means of a relatively simple program transformation whereas faceted values require modification of the runtime or explicit use of a library [32].

We present a general framework and establish its soundness with respect to explicit secrecy. Our results guarantee that the attacker learns nothing about secrets via explicit flows.

Similarly to SME, our mechanism may "repair" programs, i.e. force their security by modifying their original behavior. Yet, we show that the mechanism is precise in the sense that runs of secure programs are never modified. An example where classical taint trackers (e.g. [19]) are conservative is related to the handling of arrays. Modify the assignment in the simple example program above to be:

$$\mathbf{int}[]\ a := [0,0];\ a[h\%2] := h;\ l := a[1-h\%2]$$

This is a secure program as the value assigned to the secretly-indexed element is never used. However, a typical taint tracker would taint the entire array and raise an alarm. In contrast, our approach will accept runs of this program.

Attack detection Further, our technique can be used for attack detection. We detect attacks by matching the outcomes of the insensitive outputs from the sensitive and insensitive runs. If the values mismatch it means that there is an influence from the sensitive data to insensitive data in the original run.

In the example above, assume the default value is 0 and the secret input is 1. The detection mechanism will compare l and l' before outputting on the public sink to find out that they mismatch, being 1 and 0, respectively.

Implementation Our technique can be deployed either by extending the runtime system with shadow memories or by a source-to-source inlining transformation that injects computation on shadow memories in the original code. We implement the approach by a source-to-source transformation for an intermediate Java-like language and benchmark its precision and performance with respect to static and dynamic taint tracking. Noteworthy, language constructs such as exceptions and multithreading require no special treatment. The practical evaluation of soundness and precision uses the DroidBench [18] test suite. The results

demonstrate that performance penalty is tolerable while achieving both soundness and no false positives on the tested benchmarks.

Contributions The paper comprises these contributions: i) We present a framework of facelifted values. We illustrate the concepts for a simple imperative language with pointers and I/O (Section 2.1). ii) We establish precision results showing that runs of secure programs are never modified by the enforcement framework (Section 2.2). iii) We give a general, language-independent, view of the framework and show that our approach guarantees soundness with respect to explicit secrecy (Section 2.5). iv) We leverage our approach to build an attack detection mechanism that is free of false positives: whenever an execution is flagged by the enforcement, there is an actual attack detected (Section 2.3). v) We present DroidFace, a tool that implements our approach as a source-to-source transformation for a core of Java (Section 3). vi) We evaluate the precision and performance of DroidFace with respect to the state-of-the-art static and dynamic tools for Android applications (Section 4).

2 Facelifted Values for Taint Tracking

We present the facelifted values technique and show that it enforces explicit secrecy. To illustrate the essence of facelifted values, we introduce a simple imperative language with pointers and I/O. We briefly review explicit secrecy and show that facelifted executions enforce the property. We elaborate on the use of the enforcement technique to detect potential attacks. Lastly, we present a source-to-source transformation for statically inlining facelifted values. The proofs for lemmas and theorems can be found in the full version of the paper [5].

2.1 Language with Facelifted Values

At the heart of our mechanism is the intuition that every computation that is not related to control flow is executed multiple times, once for each security level, using default values for data from higher security levels. Consider a simple imperative language with pointers and I/O primitives:

$$
\begin{aligned}
e &::= x \mid n \mid e_1 \oplus e_2 \mid \&x \mid *e \\
c &::= \mathbf{skip} \mid c_1; c_2 \mid x \leftarrow \mathbf{alloc} \mid x := e \mid *e_1 := e_2 \\
&\quad \mid x \leftarrow \mathbf{in}(\ell) \mid \mathbf{out}(\ell, e) \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ e\ \mathbf{do}\ c
\end{aligned}
$$

The language expressions consist of global variables $x \in \mathit{Var}$, built-in values $n \in \mathit{Val}$, binary operators $\oplus$, variable references $\&x$ and dereferences $*e$. Addr is the set of memory addresses and, for simplicity, $\mathit{Addr} \subseteq \mathit{Val}$. The language constructs contain assignment, conditional, loops, input and output. In addition, the language includes dynamic memory allocation $x \leftarrow \mathbf{alloc}$ and pointer assignment $*e_1 := e_2$. We use **nil** as a way to represent uninitialized memory.

We assume a bounded lattice of security levels $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$. We write $\top$ and $\bot$ to denote the top and the bottom element of the lattice, respectively (actually a partially ordered set suffices). Each I/O channel is annotated with a fixed security label $\ell \in \mathcal{L}$. In the examples, we use a two-level security lattice consisting of **H** for variables containing confidential information and **L** for variables containing public information and $\mathbf{L} \sqsubseteq \mathbf{H}$. Input to programs is modeled by environments

mapping channels to streams of inputs values; we denote the set of environments by *Env*. Without loss of generality, we consider one stream for each level $\ell \in \mathcal{L}$. An environment $\mathcal{E} : \mathcal{L} \to \mathit{Val}^{\mathbb{N}}$ maps levels to infinite sequences of values. A facelifted value $v \in \mathit{Val}^{\mathcal{L}}$ maps levels to values; to distinguish streams and facelifted values from other functions, we write A^B for the function space $B \to A$.

We define equivalence at security level $\ell \in \mathcal{L}$ for environments, facelifted memories and traces. Intuitively, two environments, memories or traces are equivalent at level ℓ iff they look the same to an observer at level ℓ, i.e. one that can observe events at any level $\ell' \sqsubseteq \ell$, as defined by the lattice $\mathcal{L}$.

Definition 1. *Two environments $\mathcal{E}_1$ and $\mathcal{E}_2$ are ℓ-equivalent, written $\mathcal{E}_1 \approx_\ell \mathcal{E}_2$, iff $\forall \ell'.\ \ell' \sqsubseteq \ell \Rightarrow \mathcal{E}_1(\ell') = \mathcal{E}_2(\ell')$.*

A facelifted memory $m : \mathit{Addr} \to \mathit{Val}^{\mathcal{L}}$ maps addresses to facelifted values, i.e. to functions mapping levels to values. We globally fix a mapping $A(\cdot) : \mathit{Var} \to \mathit{Addr}$ from variables to addresses. Note that environments are the only source of inputs. Programs start executing with the fixed memory m_0 where $m_0(a)(\ell) = \mathbf{nil}$ for all $a \in \mathit{Addr}$ and $\ell \in \mathcal{L}$. To support pointers, we assume that $\mathit{Addr} \subseteq \mathit{Val}$. We write $m(\cdot)(\ell)$ to denote the facelifted memory projected at level ℓ, i.e. the function $a \mapsto m(a)(\ell)$. In the following, this is called ℓ-facelifted memory or non-facelifted memory (whenever the level ℓ is unimportant). We use $m, m_1, \ldots$ to range over facelifted memories and $\widetilde{m}, \widetilde{m_1}, \ldots$ to range over non-facelifted memories.

Definition 2. *Two facelifted memories m_1 and m_2 are ℓ-equivalent, written $m_1 \approx_\ell m_2$, iff $\forall \ell'.\ \ell' \sqsubseteq \ell \Rightarrow m_1(\cdot)(\ell') = m_2(\cdot)(\ell')$.*

An observation is a pair of a value and a security level, i.e. $\mathit{Obs} = \mathit{Val} \times \mathcal{L}$, or empty. We write π_1 (resp. π_2) for the first (resp. second) projection of a tuple. A trace τ is a finite sequence of observations. We write ε for the empty trace/observation. We write $\tau \restriction_\ell$ for the projection of trace τ at security level ℓ.

Definition 3. *Two traces τ_1 and τ_2 are ℓ-equivalent, written $\tau_1 \approx_\ell \tau_2$, iff $\forall \ell'.\ \ell' \sqsubseteq \ell \Rightarrow \tau_1 \restriction_{\ell'} = \tau_2 \restriction_{\ell'}$.*

We now present the semantics of the language in terms of facelifted memories and environments. We evaluate expressions in the context of a facelifted memory and, for each security level, use the memory facet of that level.

Definition 4. *The evaluation of an expression e in a facelifted memory m, written $[\![e]\!]_m \in \mathit{Val}^{\mathcal{L}}$, is defined by:*

$$
\begin{aligned}
[\![x]\!]_m(\ell) &= m(A(x))(\ell) \\
[\![n]\!]_m(\ell) &= n \\
[\![e_1 \oplus e_2]\!]_m(\ell) &= f_\oplus([\![e_1]\!]_m(\ell), [\![e_2]\!]_m(\ell)) \\
[\![\&x]\!]_m(\ell) &= A(x) \\
[\![*e]\!]_m(\ell) &= m([\![e]\!]_m(\ell))(\ell)
\end{aligned}
$$

where $f_\oplus$ denotes the semantics of the operator $\oplus$.

Figure 1 gives the operational semantics of facelifted evaluation. A state $(\mathcal{E}, m)$ is a pair of an environment $\mathcal{E}$ and a memory m. A configuration $\mathcal{E} \vdash \langle c, m\rangle$ consists of an environment $\mathcal{E}$, a command c and a memory m. We write $\mathcal{E} \vdash \langle c, m\rangle \xrightarrow{\tau}\!\!\!\!\rightarrow \mathcal{E}' \vdash \langle c', m'\rangle$ to denote that a configuration $\mathcal{E} \vdash \langle c, m\rangle$ evaluates in one step to configuration $\mathcal{E}' \vdash \langle c', m'\rangle$, producing observations $\tau \in Obs^*$. We use ε and $\lightning$ to denote normal and abnormal termination of a program, respectively.

F-Assign

$$\frac{}{\mathcal{E} \vdash \langle x := e, m\rangle \twoheadrightarrow \mathcal{E} \vdash \langle \varepsilon, m[A(x) \mapsto [\![e]\!]_m]\rangle}$$

F-IfTrue

$$\frac{[\![e]\!]_m(\top) = \mathbf{tt}}{\mathcal{E} \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, m\rangle \twoheadrightarrow \mathcal{E} \vdash \langle c_1, m\rangle}$$

F-Out

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

F-AssignPtr

$$\frac{[\![e_1]\!]_m = l \qquad l \in Addr^{\mathcal{L}} \qquad m' = update(m, l, e_2)}{\mathcal{E} \vdash \langle *e_1 := e_2, m\rangle \twoheadrightarrow \mathcal{E} \vdash \langle \varepsilon, m'\rangle}$$

F-In

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

F-Alloc

$$\frac{m' = m[A(x) \mapsto \ell \mapsto a, a \mapsto \ell \mapsto 0] \qquad a = \min\{a | a \notin \mathrm{rng}(A) \wedge \forall \ell.\ m(a)(\ell) = \mathbf{nil}\}}{\mathcal{E} \vdash \langle x \ \leftarrow \mathbf{alloc}, m\rangle \twoheadrightarrow \mathcal{E} \vdash \langle \varepsilon, m'\rangle}$$

$$\mathcal{F}_{in}(\mathcal{E}, \ell, \ell') = \begin{cases} \mathcal{E}(\ell)(0) & \ell \sqsubseteq \ell' \\ d & \ell \not\sqsubseteq \ell' \end{cases} \qquad update(m, l, e)(a')(\ell) = \begin{cases} [\![e]\!]_m(\ell) & a' = l(\ell) \\ m(a')(\ell) & \text{otherwise} \end{cases}$$

Fig. 1. Excerpt of Operational Semantics of Facelifted Evaluation

We comment on some of the interesting rules in Figure 1.The full table of rules can be found in the full version. Rule F-Assign evaluates an expression e in the context of a facelifted memory m, as in Def. 4, and yields a new facelifted memory m' where variable x is mapped to the resulting facelifted value. Similarly, rule F-AssignPtr evaluates expressions e_1 and e_2 to obtain a facelifted address and a facelifted value, respectively, and uses the function *update* to assign the latter to the former. Rule F-Alloc allocates a fresh global variable, assigns the address to x, and initializes all its facets with the default value 0. Rule F-In reads the next (non-facelifted) input value from the environment stream $\mathcal{E}$ at security level ℓ and uses the function $\mathcal{F}_{in}$ to create a facelifted value which is then assigned to global variable x. Rule F-Out evaluates an expression e in the context of the ℓ-facelifted memory and outputs the resulting (ℓ-facelifted) value to an observer at security level ℓ. Rule F-IfTrue evaluates the branch condition e *only* in the context of the $\top$-facelifted memory and executes the command in the true branch. We denote the standard evaluation relation with non-facelifted memories by $\rightarrow$; associated functions are defined analogously and reported in the full version [5].

To illustrate the facelifted semantics, consider program 1 from the introduction. Assume the domain of variables l and h is $\{0,1\}$ and the default value d is 0. The program starts executing with facelifted memory m_0 such that $m_0(A(l))(\ell) = m_0(A(h))(\ell) = \mathbf{nil}$ for all $\ell \in \mathcal{L}$. If the secret input is 1, the facelifted semantics will apply the rules F-In, F-Assign and F-Out, and output the default value 0 on the low channel. Similarly, if the secret input is 0, the output will again be 0, thus closing the leak of the insecure program. On the other hand, if we replace the output instruction in program 1 with $\mathbf{out}(\mathbf{H}, l)$, the program is secure. In fact, the variable l will be evaluated in the context of the **H**-facelifted memory and yield the correct result for an observer with high security level.

Note that declassification policies can be naturally enforced by pumping high values into low memories since the runtime has access to both during the execution (cf. Figure 1).

2.2 Explicit Secrecy

Explicit secrecy [33] is a knowledge-based security condition that formalizes the idea of "security with respect to explicit flows only". To achieve this, explicit secrecy distinguishes between changes to the state of the program and changes to the control flow, and demands security of information flows for the state part only. Concretely, it leverages the (small-step) operational semantics to extract a function that captures the state modification and the possible output for each execution step.

Definition 5. *Whenever* $\mathcal{E} \vdash \langle c, m\rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow \mathcal{E}' \vdash \langle c', m'\rangle$*, we define a function* $f : \mathit{Env} \times \mathit{Mem} \to ((\mathit{Env} \times \mathit{Mem}) \times \mathit{Obs})$ *associated with the step. For every* $\mathcal{E} \in \mathit{Env}$ *and* $m \in \mathit{Mem}$*, we define* $f((\mathcal{E}, m)) = ((\mathcal{E}', m'), \alpha)$ *where* $\mathcal{E}'$*,* m' *and* α *are unique and* $\mathcal{E} \vdash \langle c, m\rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow \mathcal{E}' \vdash \langle c', m'\rangle$*. We write* $\mathcal{E} \vdash \langle c, m\rangle \xrightarrow[f]{\alpha}\!\!\!\!\rightarrow \mathcal{E}' \vdash \langle c', m'\rangle$ *to denote this function.*

Intuitively, for each step $\mathcal{E} \vdash \langle c, m\rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow \mathcal{E}' \vdash \langle c', m'\rangle$, $f((\mathcal{E}_1, m_1))$ simulates how the input state $(\mathcal{E}_1, m_1)$ would change by executing the same step that $\mathcal{E} \vdash \langle c, m\rangle$ performs. In general, it is the language designer who defines which parts of a configuration hold state and which hold the control flow of a program. We extend the construction of state transformers to multiple steps by composing the state modifications and concatenating the output.

Example 1. Given a state $(\mathcal{E}, m)$ and command c, the state transformer for our language (cf. Def. 5) is $f((\mathcal{E}, m))$:

$$
\begin{array}{ll}
((\mathcal{E}, m[A(x) \mapsto [\![e]\!]_m]), \varepsilon) & \text{if } c = x := e \\
((\mathcal{E}, \mathit{update}(m, [\![e_1]\!]_m, e_2)), \varepsilon) & \text{if } c = *e_1 := e_2 \\
((\mathcal{E}, m[A(x) \mapsto \ell \mapsto a]), \varepsilon) & \text{if } c = x \leftarrow \mathbf{alloc} \\
\multicolumn{2}{r}{\text{where } a \notin \mathrm{rng}(A) \wedge \forall \ell.\ m(a)(\ell) = \mathbf{nil}} \\
((\mathcal{E}, m), [([\![e]\!]_m(\ell), \ell)]) & \text{if } c = \mathbf{out}(\ell, e) \\
((\mathcal{E}', m[A(x) \mapsto \ell' \mapsto \mathcal{F}_{in}(\mathcal{E}, \ell, \ell')]), \varepsilon) & \text{if } c = x \leftarrow \mathbf{in}(\ell) \\
\multicolumn{2}{r}{\text{and } \mathcal{E}' = \mathcal{E}[\ell \mapsto n \mapsto \mathcal{E}(\ell)(n+1)]} \\
((\mathcal{E}, m), \varepsilon) & \text{otherwise}
\end{array}
$$

The state transformer f acts on the memory part of the configuration (assignments, memory allocation, input and output), and leaves the control-flow statements unchanged.

We can now define the knowledge an attacker at security level ℓ obtains from observing only outputs from a sequence of changes to the state. We capture this by the set of initial environments that the attacker considers possible based on their observations. Concretely, for a given initial state $(\mathcal{E}_0, m_0)$ and a state transformer f, an environment $\mathcal{E}$ is considered possible if $\mathcal{E}_0 \approx_\ell \mathcal{E}$ and it matches the trace produced by $f((\mathcal{E}_0, m_0))$, i.e. $\pi_2(f((\mathcal{E}_0, m_0))) \approx_\ell \pi_2(f((\mathcal{E}, m_0)))$.

Definition 6 (Explicit knowledge). *The explicit knowledge at level ℓ for an environment $\mathcal{E}_0$, memory m_0 and function f, written $k_e(\ell, \mathcal{E}_0, f) \subseteq Env$, is defined by $k_e(\ell, \mathcal{E}_0, f) = \{\mathcal{E} | \mathcal{E}_0 \approx_\ell \mathcal{E} \wedge (\pi_2 \circ f)((\mathcal{E}_0, m_0)) \approx_\ell (\pi_2 \circ f)((\mathcal{E}, m_0))\}$.*

Intuitively, the attacker considers as possible all environments $\mathcal{E}$ from the set $k_e(\ell, \mathcal{E}_0, f)$. Then, given an initial state $(\mathcal{E}, m_0)$, a program satisfies explicit secrecy iff no indistinguishable initial states can be ruled out from observing the output generated by the extracted state transformer f. We define $\mathrm{id}(s) = (s, [])$.

Definition 7 (Explicit secrecy). *A program c satisfies explicit secrecy for security level ℓ and evaluation relation $\hookrightarrow$, written $ES(\hookrightarrow, \ell) \vDash c$, iff whenever $\mathcal{E} \vdash \langle c, m_0\rangle \underset{f}{\hookrightarrow}^* \mathcal{E}' \vdash \langle c', m'\rangle$, then $\forall \mathcal{E}_0.\ k_e(\ell, \mathcal{E}_0, f) = k_e(\ell, \mathcal{E}_0, \mathrm{id})$. We write $ES(\hookrightarrow) \vDash c$ iff $\forall \ell.\ ES(\hookrightarrow, \ell) \vDash c$.*

Let us consider again program 1 with the initial conditions defined as above. The program satisfies explicit secrecy for the facelifted semantics in Figure 1, i.e. $ES(\twoheadrightarrow, \ell) \vDash c_1$. Following Example 1 and Def. 5, we sequentially compose the state transformers for the input, assignment and output statements and obtain the state transformer $f((\mathcal{E}, m_0)) = ((\mathcal{E}', m'), [(0, \mathbf{L})])$, for some $\mathcal{E}'$ and m' such that $\mathcal{E} \vdash \langle c_1, m_0\rangle \twoheadrightarrow^* \mathcal{E}' \vdash \langle \varepsilon, m'\rangle$. Independently of the initial environment, f will always produce the output trace 0 for an observer at level $\mathbf{L}$. Therefore, $\forall \mathcal{E}_0.\ k_e(\mathbf{L}, \mathcal{E}_0, f) = k_e(\mathbf{L}, \mathcal{E}_0, \mathrm{id})$.

Program 1 does not satisfy explicit secrecy with respect to the standard evaluation relation. In this case, the state transformer is extracted as $f((\mathcal{E}, m_0)) = ((\mathcal{E}', m'), [(\mathcal{E}(\mathbf{H})(0), \mathbf{L})])$, capturing that the program explicitly sends the input from a high channel to a low one, thus increasing the knowledge of the observer.

The following theorems prove that facelifted execution ensures soundness and precision for any program.

Theorem 1 (Soundness). *For any program c, $ES(\twoheadrightarrow) \vDash c$.*

Theorem 2 (Precision). *If $ES(\rightarrow) \vDash c$, then $\mathcal{E} \vdash \langle c, \widetilde{m}_0\rangle \xrightarrow{\tau}{}^*$ if and only if $\mathcal{E} \vdash \langle c, m_0\rangle \xrightarrow{\tau}\twoheadrightarrow^*$.*

Explicit secrecy assumes totality on the transition relation and on the corresponding state transformers. As a result, it does not account for information leaks due to abnormal termination of a program, e.g. by applying a partial function such as division by zero. Arguably, we can consider the program

$h \leftarrow \mathbf{in}(\mathbf{H}); x := 1/h; \mathbf{out}(\mathbf{L}, 1)$ insecure since it may or may not execute the output statement depending on the input value of h, and thus leak information. We call this *crash-sensitive* explicit secrecy. Crash sensitivity can be captured by making abnormal termination visible to a low observer, e.g. by adding a special observation $\lightning$. On the other hand, leaks due to program crashes generally trigger exceptions, which can be seen as control flows, hence the program above should then be secure. We call this *crash-insensitive* explicit secrecy. Crash insensitivity can be formalized by constructing partial state transformers.

The proposed enforcement mechanism is fully precise with respect to crash-sensitive explicit secrecy, however, it may lose precision when enforcing the crash-insensitive version. We further discuss these issues in the full version [5].

2.3 Attack Detection

Theorem 2 shows that if a program is already secure, the facelifted execution produces the same outputs as the standard execution. Otherwise, the standard semantics is intentionally changed for the sake of security. Thus, a user can not tell whether or not the outcome of the computation is correct, let alone decide whether an unexpected result is due to a software bug or a security attack.

We show that facelifted semantics can be extended to detect changes to the standard program semantics and thus unveil potential attacks. Concretely, attack detection can be performed by using the following rules for output statements:

$$\text{F-OutT}\quad \frac{[\![e]\!]_m(\ell) = [\![e]\!]_m(\top)}{\mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m\rangle \xrightarrow{[([\![e]\!]_m(\ell),\ell)]}\!\!\!\!\rightarrow \mathcal{E} \vdash \langle \varepsilon, m\rangle} \qquad \text{F-OutFail}\quad \frac{[\![e]\!]_m(\ell) \neq [\![e]\!]_m(\top)}{\mathcal{E} \vdash \langle \mathbf{out}(\ell, e), m\rangle \rightarrow\!\!\!\!\rightarrow \lightning}$$

For each output statement at some security level ℓ, we evaluate the output expression both in the context of the ℓ-facelifted memory and in the context of the $\top$-facelifted memory, and compare the results. Since the $\top$-facelifted memory is never affected by the default values, it will always contain the result of evaluation under the standard semantics. Therefore, if the two values differ, we have detected an attack and thus terminate the execution abnormally. The following theorem shows that the abnormal termination implies real attacks (only true positives).

Theorem 3 (Attack Detection). *If* $\mathcal{E} \vdash \langle c, m_0\rangle \rightarrow\!\!\!\!\rightarrow^* \lightning$, *then* $ES(\rightarrow) \not\vdash c$.

Like SME [17] and faceted execution [2], the mechanism can fail to detect insecurities, i.e. $ES(\rightarrow) \not\vdash c \not\Rightarrow (\forall \mathcal{E}.\ \mathcal{E} \vdash \langle c, m_0\rangle \rightarrow\!\!\!\!\rightarrow^* \lightning)$. This happens if the the chosen default values produce the same outputs as the real execution, for example if the default values are equal to the real inputs. Even if the program is run with multiple different default values, this may not be detected (consider, for example the program $h \leftarrow \mathbf{in}(\mathbf{H}); \mathbf{out}(\mathbf{L}, (h - d_1) \times (h - d_2))$ where d_1 and d_2 are possible default values. For an environment $\mathcal{E}$ where $\mathcal{E}(\mathbf{H})(0) \in \{d_1, d_2\}$, the above detection will never see the attack, despite the program being insecure. More generally, trying to detect an attack using a finite set $\mathcal{D}$ of default values will yield a false negative if the high input matches any of the default values; then the following program will hide an insecurity: $h \leftarrow \mathbf{in}(\mathbf{H})$; $\mathbf{out}(\mathbf{L}, \Pi_{d \in \mathcal{D}}(h - d))$.

In practice, random defaults or multiple defaults for a single location take us a long way. We obtain no false negatives on the DroidBench suite, as reported in Section 4.

This also entails that despite the precision and soundness results, this mechanism does not give rise to a decision procedure for (per-run) explicit secrecy.

2.4 Inlining Facelifted Values through Static Program Transformation

Facelifted evaluation enforces explicit secrecy dynamically by means of unconventional semantics, as described in Figure 1. This requires modification of the underlying language runtime which makes it difficult to deploy for many settings. We present a program transformation that statically inlines facelifted values into the source code and uses standard semantics to achieve the same result as facelifted evaluation. We transform a program $c \in \mathit{Com}$ by applying a transformation function $\mathcal{T}(\cdot) : \mathit{Com} \to \mathit{Com}$. For each security level $\ell \in \mathcal{L}$ and for each variable x, we introduce a shadow variable x_ℓ to carry the ℓ-facelifted value for an observer at level ℓ. We write $[e]_\ell$ to denote the renaming of all variables x in e with x_ℓ and $;S$ to denote the sequential composition of commands from a set S.

$$
\begin{array}{ll}
\mathcal{T}(\textbf{skip}) & = \textbf{skip} \\
\mathcal{T}(x \leftarrow \textbf{alloc}) & = ; \{[x]_\ell \leftarrow \textbf{alloc} \mid \ell \in \mathcal{L}\} \\
\mathcal{T}(x := e) & = ; \{x_\ell := [e]_\ell \mid \ell \in \mathcal{L}\} \\
\mathcal{T}(*e_1 := e_2) & = ; \{*[e_1]_\ell := [e_2]_\ell \mid \ell \in \mathcal{L}\} \\
\mathcal{T}(\textbf{out}(\ell, e)) & = \textbf{out}(\ell, [e]_\ell) \\
\mathcal{T}(x \leftarrow \textbf{in}(\ell)) & = x_\ell \leftarrow \textbf{in}(\ell);\ ; \{\mathcal{T}_{in}(\ell, \ell') \mid \ell' \in \mathcal{L} \text{ and } \ell \neq \ell'\} \\
\mathcal{T}(c_1 \,;\, c_2) & = \mathcal{T}(c_1) \,;\, \mathcal{T}(c_2) \\
\mathcal{T}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) & = \textbf{if } [e]_\top \textbf{ then } \mathcal{T}(c_1) \textbf{ else } \mathcal{T}(c_2) \\
\mathcal{T}(\textbf{while } e \textbf{ do } c) & = \textbf{while } [e]_\top \textbf{ do } \mathcal{T}(c)
\end{array}
$$

where $\mathcal{T}_{in}(\ell, \ell')$ equals $x_{\ell'} := x_\ell$ if $\ell \sqsubseteq \ell'$, otherwise $x_{\ell'} := d$.

Note that faceted values and SME can not be implemented as easily with a program transformation [4]. We then show the correctness of the transformation.

Theorem 4 (Correctness). $\mathcal{E} \vdash \langle c, m_0 \rangle \overset{\tau}{\twoheadrightarrow}^* $ *iff* $\mathcal{E} \vdash \langle \mathcal{T}(c), \widetilde{m}_0 \rangle \overset{\tau}{\to}^*$.

Corollary 1 (Soundness and Precision). $\mathcal{T}(c)$ *is sound and precise.*

2.5 General Framework

The presented approach is not specific to a concrete language. The full version [5] presents a general version of this technique applicable to a wide range of languages under fairly unrestrictive assumptions: The number and level of outputs performed by an evaluation step may not depend on the memory; moreover, the semantics is assumed to be total and deterministic. Under these assumptions, satisfied by many realistic languages, the framework provides soundness and precision guarantees. Moreover, we also sketch an approach to lift the totality assumption on the semantics.

3 Implementation

This section presents DroidFace, a dynamic analysis tool for taint tracking in Android applications, based on the facelifted values from Section 2. The tool is a prototype built on top of the *Soot* framework [37]. DroidFace leverages an intermediate bytecode language, *Jimple* [37], to implement the static source-to-source transformation for facelifted evaluation. As a result, the implementation works with both ordinary Java *class* files as well as *APK* files for the Android platform. We further discuss Jimple in the full version of the paper [5].

DroidFace We give a general overview of the architecture, features and limitations of DroidFace. We emphasize that the main contribution is the development of a fundamentally new approach to taint tracking applicable in many settings. Our main goal is to demonstrate feasibility of our approach in terms of precision, performance and flexibility, *not* to fully cover the Android platform.

DroidFace takes as input an Android APK file (a compressed archive) and uses Soot to convert it to a set of Jimple programs. Next, it applies the source-to-source transformation (as outlined in Section 2.4) to inline facelifted values and therefore produce a secure version of the input program. Finally, DroidFace converts the program back to an APK file that can be run on the Android platform. The source code of DroidFace is available online [5].

DroidFace is implemented in *Scala* [29] and supports an arbitrary lattice, represented as a Scala class. Noteworthy, many language constructs such as exceptions and multithreading require no special treatment and are transformed correctly by DroidFace. Control-flow statements like **if** e **goto** pc are transformed to refer to the variable copies at level $\top$. Similarly, method invocations with **virtualinvoke** may select an object based on secret data, resulting in a control-flow leak; as an example, consider a program allocating two objects of type A and calling a method that sends the first constructor argument:

$A[]\ x = [\mathbf{new}\ A(1), \mathbf{new}\ A(2)]; x[h\%2].send()$

This is a leak through the program's control-flow (execution jumps to a different object's method depending on h), hence we use the values at level $\top$.

Since secret input usually consists of primitive data, such as numbers or strings, the transformation only replicates variables and fields of primitive types. Moreover, this is needed to avoid duplicating calls to constructors and other methods, as they may have side effects that should only be performed once. Since bodies of built-in methods are not affected by the program transformation, such calls need to be handled specially. Calls to side-effect free methods, e.g. `java.lang.Math.sin()`, can be duplicated for each level. However, other methods, e.g. sending data over the network, must only be performed once. A whitelist is used to determine which methods are side-effect free.

The implementation makes a number of simplifications. For example, file access is not handled in a precise way: While an implementation could duplicate file contents to maintain both soundness and precision, doing so in an efficient manner would require deduplication to manage the storage space overhead. As a result, the implementation writes either the *low* or *high* data to a file, depending on a configuration parameter. Inter-application communication (IAC) has not been modified to propagate facelifted values. However, the approach extends naturally to IAC by adding data for all levels to objects passed between apps.

Facelifted values are passed between methods by creating objects that contain one field for each level in the lattice. These objects are constructed for each primitive argument at a call site and returned by each non-entry method with a primitive return type. This creates additional overhead due to object creation; however, this can be avoided if facelifted values are implemented by modifying the runtime. More details on the performance impact can be found in Section 4.

Since source and sink detection is covered in related work [21,1], we use an incomplete set of known sources and sinks for the purposes of this evaluation.

Alternative strategies While implementing facelifted values via program transformation as presented here provides a reasonable proof-of-concept implementation, there are a number of alternative techniques that can be explored. A minor optimization is to avoid creating a new object when passing facelifted values to methods; however, this is still necessary when passing facelifted return values. One possible approach is to simply run the program twice, once with real values while recording control-flow decisions and once with default values making use of the recorded control flow. However, this requires careful suppression of publicly observable side effects and outputs in the run with real values and vice versa. Moreover, this technique requires synchronization of the two runs of the program, leading to similar issues as SME [17].

4 Benchmarks

This section evaluates our prototype. Soundness and precision are evaluated using the *DroidBench* [1] benchmark suite. DroidBench is a set of small Android apps to evaluate static analysis tools for the Android platform. The main goal is to test sensitivity of a static analysis with respect to complex language features. To obtain better coverage for dynamic analysis, we have developed additional micro-applications that exercise other features such as path sensitivity and complex expression evaluation. As described in Section 3, the implementation does not support the full range of Android features; as a result we only provide partial benchmark results. However, the current results indicate that the presented approach prevents information leaks while not producing false positives. For performance evaluation, we use the *CaffeineMark* [8] benchmark suite to compare our implementation to both, unmodified Android, as well as *TaintDroid* [19].

Precision We run DroidFace on a number of examples from DroidBench. Due to constraints outlined in Section 3, not all examples are used for this evaluation. Moreover, a number of examples, such as emulator detection, are not relevant to a dynamic enforcement technique. Furthermore, some examples produced errors (e.g. missing permissions) when executed and could not be tested.

For the tested examples, DroidFace remains both sound and precise. A more detailed comparison to other taint-tracking systems can be found in the full version. Note that TaintDroid also maintains soundness and precision for all tested APKs (with the exception of `PublicAPIField1.apk` for which TaintDroid is unsound). However, TaintDroid does not remain fully precise in the presence of arrays. Consider the following secure program similar to the example from Section 1:

$$\mathbf{int}[]\ a := [0,0];\ a[h\%2] \leftarrow \mathbf{in}(\mathbf{H});\ \mathbf{out}(\mathbf{L}, a[1-h\%2])$$

Since a secret value h is assigned to a position in an array that depends on a secret, TaintDroid taints the entire array a and hence yields a false positive. DroidFace, however, produces the unmodified trace.

Performance We compare the performance of DroidFace to TaintDroid and unmodified Android using the CaffeineMark benchmark suite. For running the benchmark on Android, we used a CaffeineMark app [9] from Google Play. Figure 2 shows a comparative performance evaluation using an ARM-based Android emulator running Android 4.3. The emulator was run on a Dell Latitude E7440 laptop with a i7-4600U CPU. Performance benchmarks on an emulator are indicative at best, but we did find that the results reported by CaffeineMark stabilize after running the benchmarks a few times to allow for startup effects to die out. The figure shows the scores of the fifth run of the benchmark.

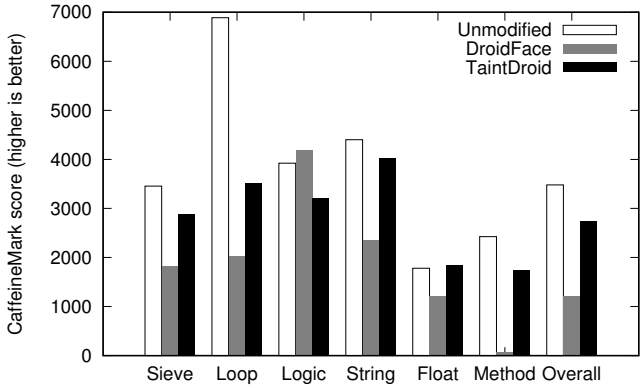

Fig. 2. Performance Comparison using *CaffeineMark*

CaffeineMark reports a custom score value useful for relative comparison only. The scores for individual categories are proportional to the number of times a test is run divided by the average time of a test execution. The *Overall* score is the geometric mean of the scores in the individual categories. A description of the individual categories can be found in the full version.

The performance overhead is not prohibitively high given that DroidFace, being a proof-of-concept, produces unoptimized code. Note that due to different experimental setup and different versions of TaintDroid and CaffeineMark, our measurements for TaintDroid differ from the previously reported results [19]. Also note that many popular applications are not bound by CPU performance, but by user interaction[19]; hence, the real-world impact of these performance results may be negligible. The increase in code size is minor, as the bytecode itself is probably much smaller than other resources that ship with Android application. A more detailed comparison can be found in the full version. Similarly, preliminary experiments show the increase in memory usage to be insignificant as well, as the memory required for duplicates of primitive values is overshadowed by other resources loaded by the application.

5 Related Work

This section compares our work with closely related works for Android security. Table 1 gives a comparative overview of the state-of-the-art (static and dynamic)

approaches to enforcing confidentiality for Android apps. We elaborate on the data from Table 1 (on a scale from ✗, 1, 2 to ✓) and other related work.

Reasoning about Taint Tracking. Taint tracking has been widely adopted as an ad-hoc technique in many security settings both for confidentiality and integrity. As a result, the majority of existing works either propose taint tracking as a bug-finding tool or justify its correctness using informal and custom-tailored soundness criteria [12,26,1,19,21,10]. Recently, Schoepe et al. [33] have proposed *explicit secrecy*, a semantic characterization of policies for taint tracking that captures the intuition of tracking direct data dependencies and generalizes Volpano's *weak secrecy* [39].

Table 1. Comparison of State-of-the-art Taint Trackers

Tool	Value	Flow	Context	Object	Path	Arrays	Native	Enforcement	Sound	Precise
FlowDroid	✗	✓	✓	✓	✗	✗	1	Static	✗	✗
Amandroid	✓	✓	✓	✓	✗	✗	1	Static	✗	✗
DroidSafe	✗	✗	✓	✓	✗	✗	1	Static	✗	✗
HornDroid	✓	1	✓	✓	✗	1	1	Static	1	✗
TaintDroid	✓	✓	✓	✓	✓	1	1	Dynamic	✗	✗
AppFence	✓	✓	✓	✓	✓	✗	1	Dynamic	✗	✗
DroidFace	✓	✓	✓	✓	✓	✓	2	Dynamic	✓	✓

Our work presents a general technique for precise enforcement of explicit secrecy through facelifted execution and establishes soundness and precision. As reported in Table 1 (SOUND), the majority of existing approaches use taint-tracking in an ad-hoc manner without providing formal security justifications (✗). It is noteworthy that HornDroid [10] presents a correctness proof for their abstract data-flow analysis with respect to an instrumented semantics of Android activities. Instrumented semantics allow to approximate non-safety security properties such as explicit secrecy in terms of safety properties, thus not capturing the precise semantics of taint tracking. Chaudhuri et al. [12] and Livshits and Chong [26] propose similar conditions. Our enforcement is the first fully precise mechanism with respect to explicit secrecy (no false positives). As discussed before, existing approaches differ in precision, yet none of them is fully precise.

Static Taint Analysis. Static analysis has been proposed to tracking explicit flows in the Android domain. As shown in Table 1, static approaches differ on the features of analysis they implement. The complexity of the Android stack makes static analysis cumbersome and often leads to unsoundness or false positives [21].

Bodden et al. [1] present FlowDroid, a static taint analysis tool for Android applications. The analysis is path- and value- insensitive and it overapproximates arrays in a coarse manner. Due to event-driven nature of Android apps (multiple entry point, asynchronous components, callbacks, $\cdots$), flow sensitivity often leads to incompleteness and false negatives [21]. Li et al. [25] present IccTA, an extension of FlowDroid analysis with inter-component communication. Gordon et al. present DroidSafe [21], a static taint analysis tool that offers a high degree of precision and soundness. DroidSafe is a significant engineering effort to model the Android runtime behavior for non-Java code (native methods, event callbacks, component lifecycle, hidden state) using analysis stubs. The analysis

is flow insensitive since interactions between apps and the Android environment are mediated by asynchronous callbacks, hence all event orderings are to be considered. Points-to and information flow analysis are also flow insensitive, which improves scalability at the expense of losing precision. DroidSafe may raise false positives due to the coarse modeling of arrays, maps, lists, flow insensitivity or event ordering. It is worth noting that although DroidSafe is more precise than FlowDroid, yet the number of false positives is too high for unknown applications. Wei et al. [40] present Amandroid, a general framework for determining points-to information for all objects in Android apps in a flow and context-sensitive way across Android apps components. Amandroid can be specialized to solve security problems including taint tracking with high precision.

Calzavara et al. [10] present HornDroid, an approach that uses Horn clauses for soundly abstracting the semantics of Android applications. HornDroid expresses security properties as a set of proof obligations that are automatically discharged by an off-the-shelf SMT solver. The analysis is value- and context-sensitive. The authors argue that these features in combination are important. The analysis is flow sensitive on registers and flow insensitive on heap locations and callback methods, which increases precision without compromising soundness. Moreover, the static analysis is field-insensitive on arrays, although, being value-sensitive, HornDroid supports a more precise treatment of array indexes.

Static analysis approaches have the advantage of no-runtime overhead, however, especially for Android, they are fated to be imprecise. This is not only due to the complexity of the language and the execution lifecycle, but also to theoretical and practical limitations of current verification technologies. For instance, DroidSafe uses analysis stubs for native methods to approximate the data flow, object instantiation and aliasing of the missing native code. As recognized by the authors, this approach is not always sound. DroidFace faces the same problems with respect to native code with side effects, yet being fully precise for side-effect free native code. On the downside, DroidFace introduces runtime overhead which may deteriorate the performance.

Dynamic Taint Analysis. Dynamic taint analysis has been proposed for tracking privacy leaks at runtime in Android applications. Enck et al. present TaintDroid [19] a system-wide integration of dynamic taint tracking into Android. TaintDroid simultaneously tracks sensitive data from multiple sources by extending and modifying the Android environment with taint tags. Tracking is done at different levels: (i) variable-level: by instrumenting the Dalvik VM; (ii) message-level for IPC communication: by assigning one taint per serialized object (parcel); (iii) method-level: by providing method summaries for native methods; (iv) file-level: by assigning one taint per file. TaintDroid has different sources of false positives: for instance, by assigning one taint per file or one taint per parcel. Surprisingly, we found out that although the paper claims to assign one taint per array [19], the TaintDroid tool appears to assign one taint per array cell in our experiments. Native methods take the union of arguments' taints as resulting taint for the method return, which may cause false negatives due to possible side effects. TaintDroid requires modification of the JIT compiler in order to implement the taint propagation logic for applications using JIT. By contrast, our source-to source transformation requires no modification of the Android runtime

and it is more precise. Hornyack et al. present AppFence [22], an extension of TaintDroid with shadow data for sensitive information that a user does not want to share. As for TaintDroid, AppFence implements modifications at the Android OS level. In addition to the precision issues inherited by TaintDroid, AppFence modifies the semantics of secure apps that never leak sensitive information.

Beyond Taint Tracking. DroidFace does not prevent insecure information flows through *covert channels*. For instance, applications can still leak sensitive information through implicit flows [16], where the information may flow through the control structure of the program. Information flow control [31] comprises methods and techniques that, in addition to explicit flows, also prevent implicit flows. Soundness is typically shown with respect to the semantic property of noninterference [20]. Information flow security has been explored in the context of Android apps by, e.g., Lortz et al. [27] and Jia et al. [23].

Our work draws inspiration from SME [17]. SME provides a precise enforcement of noninterference by running a program in a controlled manner, once for each security level. Kashyap et al. [24] study different scheduling strategies for SME and address the subtleties of timing- and termination-sensitive noninterference. Rafnsson and Sabelfeld [30], and Zanarini et al. [42] explore scheduling strategies with the goal to leverage SME for attack detection. By contrast to SME, our enforcement does not require scheduling different program copies.

Austin and Flanagan [2,32] enforce noninterference by runtime manipulation of faceted values. A challenging aspect for this line of work is dealing with nonlocal control flow and I/O, as the facets must record what can happen when the program takes different control-flow paths. Having explicit secrecy as the goal, our approach is free of these challenges because under our enforcement the program takes the same control-flow path as the original run. Section 1 offers further points of contrast to facelifted values.

Barthe et al. [4] implement SME for noninterference through static program transformation. This approach inherits the scheduling issues from SME and requires the buffering of inputs from lower security levels so that these inputs can be reused by executions running at higher security levels. These issues are not present in our work at the expense of enforcing the more liberal security policy.

Jeeves [41] is a programming language that uses symbolic evaluation and constraint-solving for enforcing information-flow policies. Austin et al. [3] show how to extend Jeeves with faceted values to propagate multiple views of sensitive information in a single faceted execution.

Boloşteanu and Garg propose asymmetric SME with declassification [7], focusing on robustness of SME wrt. modified inputs. This is achieved by producing a *low slice*, a program to compute the public results of the original program.

6 Conclusion

We have presented a dynamic mechanism for taint tracking. Its distinguishing feature is that it does not track taint propagation. Instead, it duplicates the state, representing its tainted and untainted views. We have showed that the mechanism is sound with respect to the policy of explicit secrecy and that it is precise in the sense that runs of secure programs are never modified. Further, we have leveraged the mechanism to detect attacks with zero false positives:

whenever a mismatch between tainted and untainted views is detected, it must be due to an attack. Finally, we have implemented DroidFace, a source-to-source transformation for an intermediate Java-like language and benchmarked its precision and performance with respect to typical static and dynamic taint trackers for Android apps. The results show that performance penalty is tolerable while achieving both soundness and no false positives on the tested benchmarks.

Future work includes support for declassification policies. Recent progress on declassification for SME [2,30,38,7] gives us an encouraging start. Exploring facelifted values for machine code integrity is another promising line of future work. We are also interested in extending and optimizing the DroidFace tool as to make it suitable for a large-scale study of Android apps from Google Play. Finally, we will also explore memory optimizations in cases of large numbers of security levels, avoiding duplication.

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

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI (2014)
2. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: POPL (2012)
3. Austin, T.H., Yang, J., Flanagan, C., Solar-Lezama, A.: Faceted execution of policy-agnostic programs. In: PLAS (2013)
4. Barthe, G., Crespo, J.M., Devriese, D., Piessens, F., Rivas, E.: Secure multi-execution through static program transformation. In: FMOODS/FORTE (2012)
5. Let's face it: Faceted values for taint tracking. Full version and implementation. `http://www.cse.chalmers.se/research/group/security/facets`
6. Bielova, N., Rezk, T.: A taxonomy of information flow monitors. In: POST (2016)
7. Boloşteanu, I., Garg, D.: Asymmetric secure multi-execution with declassification. In: POST (2016)
8. Caffeinemark. `http://www.benchmarkhq.ru/cm30/`
9. Caffeinemark for android. `https://play.google.com/store/apps/details?id=com.android.cm3`
10. Calzavara, S., Grishchenko, I., Maffei, M.: Horndroid: Practical and sound security static analysis of android applications by smt solving. In: EuroS&P (2016)
11. Capizzi, R., Longo, A., Venkatakrishnan, V.N., Sistla, A.P.: Preventing information leaks through shadow executions. In: ACSAC (2008)
12. Chaudhuri, A., Naldurg, P., Rajamani, S.K.: A type system for data-flow integrity on windows vista. PLAS (2008)
13. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: TaintTrace: Efficient flow tracing with dynamic binary rewriting. In: ISCC (2006)
14. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: USENIX Security Symposium (2004)
15. Cohen, E.S.: Information transmission in sequential programs. In: FSC. Academic Press (1978)
16. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM (1977)
17. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: S&P (2010)

18. Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for android. https://github.com/secure-software-engineering/DroidBench
19. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (2014)
20. Goguen, J.A., Meseguer, J.: Security policies and security models. In: S&P (1982)
21. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS (2015)
22. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In: CCS (2011)
23. Jia, L., Aljuraidan, J., Fragkaki, E., Bauer, L., Stroucken, M., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on android - (extended abstract). In: ESORICS (2013)
24. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and termination-sensitive secure information flow: Exploring a new approach. In: S&P (2011)
25. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: Iccta: Detecting inter-component privacy leaks in android apps. In: ICSE (1) (2015)
26. Livshits, B., Chong, S.: Towards fully automatic placement of security sanitizers and declassifiers. In: POPL (2013)
27. Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: Towards a Certifying App Store for Android. In: SPSM (2014)
28. Netscape: Using data tainting for security. http://www.aisystech.com/resources/advtopic.htm (2006)
29. Odersky, M., Rompf, T.: Unifying functional and object-oriented programming with scala. Commun. ACM 57(4) (2014)
30. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: Fine-grained, declassification-aware, and transparent. In: CSF (2013)
31. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. JSAC (2003)
32. Schmitz, T., Rhodes, D., Austin, T.H., Knowles, K., Flanagan, C.: Faceted dynamic information flow via control and data monads. In: POST (2016)
33. Schoepe, D., Balliu, M., Pierce, B.C., Sabelfeld, A.: Explicit secrecy: A policy for taint tracking. In: EuroS&P (2016)
34. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: S&P 2010 (2010)
35. Song, D.X., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: ICISS (2008)
36. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: PLDI (2009)
37. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: CASCON (1999)
38. Vanhoef, M., De Groef, W., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: CSF (2014)
39. Volpano, D.M.: Safety versus secrecy. In: SAS (1999)
40. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: CCS (2014)
41. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. In: POPL (2012)
42. Zanarini, D., Jaskelioff, M., Russo, A.: Precise enforcement of confidentiality for reactive systems. In: CSF (2013)