TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies*

Mads Dam
KTH Royal Institute of Technology
Stockholm, Sweden
mfd@kth.se

Gurvan Le Guernic
KTH Royal Institute of Technology
Stockholm, Sweden
gurvan@kth.se

Andreas Lundblad
KTH Royal Institute of Technology
Stockholm, Sweden
landreas@kth.se

ABSTRACT

Current approaches to security policy monitoring are based on linear control flow constraints such as `runQuery` *may be evaluated only after* `sanitize`. However, realistic security policies must be able to conveniently capture data flow constraints as well. An example is a policy stating that *arguments to the function* `runQuery` *must be either constants, outputs of a function* `sanitize`*, or concatenations of any such values.*

We present a novel approach to security policy monitoring that uses tree automata to capture constraints on the way data is processed along an execution. We present a λ-calculus based model of the framework, investigate some of the models meta-properties, and show how it can be implemented using labels corresponding to automaton states to reflect the computational histories of each data item. We show how a standard denotational semantics induces the expected monitoring regime on a simple "while" language. Finally we implement the framework for the Dalvik VM using TaintDroid as the underlying data flow tracking mechanism, and evaluate its functionality and performance on five case studies.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging—*Monitors*; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

Keywords

Runtime monitoring, policy enforcement, tree automata

1. INTRODUCTION

Today 95% of all mobile devices run Android, Symbian, iOS or RIM [15]. All those OS share the same security model for third party applications. When a new application is installed (or launched for the first time) the operating system asks the user if he or she grants the application a set of permissions. Such permissions typically allow the application to access internet, the GPS hardware, address book data, camera, etc. Unfortunately the model is quite crude. Most useful and innocent tasks require a combination of permissions which could just as well be used maliciously [12, 35]. Many applications request the Internet access permission, for example in order to display ads, together with permissions for other phone resources, which can then potentially be remotely accessed and controlled. For this reason it is of high importance to study techniques, such as the one proposed in this paper, which allow policies to be expressed at a finer level of granularity.

Our proposal is to use bottom-up tree automata to track how an application processes data at runtime. The approach monitors how each data item in an execution has been computed and prevents certain function calls from being made based on this information. This data-centric approach to runtime monitoring allows for a wide range of policies to be expressed, including API usage policies restricting which methods may be applied to what arguments and data flow policies stating how data must have been processed before being passed to certain functions.

The policies in this framework are different from the ones handled by existing techniques. For example, as opposed to existing runtime monitoring techniques which handle policies expressing temporal properties such as "*f may be invoked after g has been invoked but not vice versa*" our approach handles policies such as "*f may be applied to the result of g but not vice versa*". A more concrete example of a policy which is naturally expressed in our framework (but difficult or impossible to express in others) could for instance state that `sanitize` accepts any string as argument, while the function `runQuery` only accepts string constants, strings returned by `sanitize` or concatenations of such strings.

The approach described in this paper differs from traditional runtime monitoring on three key points. The approach is (a) data centric, (b) based on tree shaped traces and (c) relies on richer observable actions. (a) Standard techniques [11, 20, 13, 36] are control flow oriented: They monitor the linear flow of events as they occur at system/thread/object level. By contrast, our technique is data oriented, allowing different flows of data to be monitored in isolation, even if they are arbitrarily interleaved in the application. (b) Existing runtime monitoring frameworks are typically based on deterministic finite automata (DFA) [8,

*Work partially supported by the EU FP7 project HATS, by the Swedish Strategic Research Foundation project PROSPER, and by the ACCESS Linnaeus Centre at KTH.

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

11, 20], edit automata [22], LTL [27, 30], context free grammars [25], or a variation thereof [23, 16], all of which rely on a model of *linear* traces. Since our approach focuses on how data is processed, i.e. how functions are combined rather than in what order actions are performed, traces manipulated in our framework are tree shaped. (c) Similarly to work by others [2, 20, 19, 25, 22], we let the function calls be the actions observable by the monitor. However, the fact that monitoring is performed at the data level allows the observable actions to depend on the computational history of each arguments in a manner which is impossible or inconvenient using the existing frameworks.

The first contribution of the paper is a theoretical formalization of the framework using λ-calculus. It includes a program model which records computational histories of data, and a policy model which accepts or rejects certain computations. As our second contribution we identify three policy classes and establish their relationships. As a third contribution, the paper describes a solution allowing an efficient implementation of the approach by using so called *labels* which are to be seen as abstractions of computational histories. We show how policies, which are semantically defined in terms of bottom-up tree automata, can be enforced using data labels corresponding to the automaton states. As a validation of our framework we then show how a standard denotational semantics induces the expected monitoring regime on a simple imperative language. The final contribution is an implementation of the framework for Java programs run on top of the Android platform. The implementation relies on *taint tracking* for its underlying data flow mechanism and on *monitor inlining* for policy enforcement. The practicality of the approach is demonstrated in five different case studies.

The theoretical part of the paper is related to the work on labeled λ-calculus which was initially proposed by Lévy [21]. A labeled λ-calculus associates labels with subterms in order to track how they affect the reduction. Gandhe, Venkatesh and Amitabha [14] use this as a theoretical basis for analysis of certain aspects of functional programs. Specifically, they define a notion of *need* and show how to use the calculus to identify to what extent an argument is needed to reduce a function application to its head normal form. This involves tracking computations and origins of subterms just as required by our framework. However, the existing labeled λ-calculi do not reflect the exact semantics of practical data flow tracking techniques such as the taint tracking mechanism on which our framework relies, which is why the calculus presented in this paper differs from existing ones.

Taint analysis is a well known technique for tracking direct data flows and has been studied extensively over the years. Our framework is built upon the *TaintDroid* taint analysis framework [10]. TaintDroid targets Android applications and is based on an extension of the Dalvik VM. The extension allows for simultaneous real-time tracking of data coming from multiple different sources with the relatively small runtime overhead of 14%.

The inlining algorithm presented in the paper builds upon the algorithm described by Dam et al [8, 7] with important differences regarding the representation and manipulation of automaton states.

Several papers describe *static* approaches for checking and enforcing policies related to the ones handled in our framework. These approaches usually rely on some form of type system and typically focuses on checking API protocols. The programming language concept of *typestates* is one example of such an approach. Typestate is a refinement of the concept of a type: whereas the type of an object determines the set of operations *ever* permitted on the object, a typestate determines the subset of these operations which is permitted in a particular context. The idea was introduced by Strom and Yemini [31] and has recently been developed further by DeLine and Fändrich [9] and by Bierhoff and Aldrich [2, 3]. When compared to our approach, a typestate could be seen as the compile-time counterpart of a label. However, just as the runtime type of an object is more precise than its static type, our dynamic labels are more precise than typestates. The higher precision available at runtime allows us to avoid many of the problems that static program analysis faces due to, for instance, aliasing and concurrency. Furthermore, even if typestates were tracked and inspected in runtime, our notion of label is more general than typestates, since labels are not bound to a specific type and since labels can propagate from one object to another.

2. A CALCULUS WITH API FUNCTIONS

This section presents the calculus used as theoretical foundation. The calculus is an untyped λ-calculus extended with constants, ranged over by $c \in \mathsf{C}$, n:ary function symbols, ranged over by $f^n \in \mathsf{F}$ and choice, written $(t = t)\ t\ t$. Applying f^n on $c_1, \ldots, c_n$ yields a value in C atomically and without side-effects. The semantics of functions is externally defined and written simply as $[\![f^n(c_1, \ldots, c_n)]\!]$. The calculus grammar follows:

$$t ::= v \mid t\ t \mid (t = t)\ t\ t$$
$$v ::= x \mid c \mid \lambda x\,.\,t \mid f^n\ c_1 \ldots c_m \qquad \text{where } m < n$$

The standard transition relation $\rightharpoonup$ is identical to the extended transition relation $\rightarrow$ of Figure 1 with the $\hat{\tau}$-related annotations removed.

We regard terms as programs, and finite (resp. infinite) sequences of reductions, written $t_0 \rightharpoonup t_1 \rightharpoonup \ldots \rightharpoonup t_n$ (resp. $t_0 \rightharpoonup t_1 \rightharpoonup \ldots$), as *runs* or *executions*. For brevity, we write $t_0 \rightharpoonup t_1 \rightharpoonup \ldots (\rightharpoonup t_n)$ when our reasoning applies to both finite and infinite executions.

Example 1. *Provided* $[\![\texttt{userInput}()]\!]$ *yields some string* s, $[\![\texttt{flipCoin}()]\!]$ *yields either* hd *or* tl, $[\![\texttt{sanitize}(s)]\!]$ *yields* s', *and* $[\![\texttt{exec}(c)]\!]$ *yields* r_c, *the following program:*

$$\texttt{exec}\ (\lambda x\,.\,((\texttt{flipCoin} = \mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ \texttt{userInput})$$

executes as follows whenever $[\![\texttt{flipCoin}()]\!]$ *yields* hd*:*

$\rightharpoonup \texttt{exec}\ (\lambda x.((\texttt{flipCoin} = \mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ s)$
$\rightharpoonup \texttt{exec}\ ((\texttt{flipCoin} = \mathsf{hd})\ (\texttt{sanitize}\ s)\ s)$
$\rightharpoonup \texttt{exec}\ ((\mathsf{hd} = \mathsf{hd})\ (\texttt{sanitize}\ s)\ s)$
$\rightharpoonup \texttt{exec}\ (\texttt{sanitize}\ s)$
$\rightharpoonup \texttt{exec}\ s'$
$\rightharpoonup \mathsf{r}\_s'$

This execution is safe as the user input is sanitized before being executed. If $[\![\texttt{flipCoin}()]\!]$ *yields* tl*, the execution proceeds as follows:*

$\ldots$
$\rightharpoonup \texttt{exec}\ ((\mathsf{tl} = \mathsf{hd})\ (\texttt{sanitize}\ s)\ s)$
$\rightharpoonup \texttt{exec}\ s$
$\rightharpoonup \mathsf{r}\_s$

This execution is unsafe. As the user input is not sanitized, the user can execute any "bad" command.

Example 1 emphasizes a first distinction between static verification and our dynamic approach. Static techniques reject the whole program as it contains at least one bad execution. Our approach rejects executions where $[\![\texttt{flipCoin}()]\!]$ yielded tl, but accepts the others.

2.1 Observable Actions

An *observable action* is an action performed by the program, and observed by the execution monitor (and possibly rejected). As in similar work [7, 8, 20, 19, 11], the observable actions are the calls to external functions. However, the novelty is the fact that not only function identifier and argument values are taken into account, but also the history of how the arguments were computed.

To keep track of the history of the computations the resulting constants are annotated with a *function application tree* (FAT), $\hat{\tau} ::= f(\tau_1, \ldots, \tau_n)$ where $\tau ::= \hat{\tau} \mid c$. A FAT is intended to capture the full history of function applications producing a constant. For example, $\mathsf{f}(\mathsf{g}()) : \mathsf{1}$ means that $\mathsf{1}$ is the result of applying $\mathsf{f}(\cdot)$ to $\mathsf{g}()$. Each time a function f^n is called with some arguments, $\tau_1 : c_1, \ldots, \tau_\mathsf{n} : c_n$, a new tree is constructed: $f(\tau_1, \ldots, \tau_n)$. This newly created tree serves both as the annotation of the resulting constant, and as the descriptor of the observable action that took place. The reduction of t to t' is written $t \xrightarrow{\hat{\tau}} t'$ if it generates the observation $\hat{\tau}$, and $t \xrightarrow{\epsilon} t'$ otherwise. The extended semantics with annotations is described in Figure 1. An unannotated reduction sequence can always be annotated to form a corresponding annotated reduction sequence, and vice versa. In other words, $\rightharpoonup$ and $\rightarrow$ are bisimilar. Depending on the context, $[\hat{\tau}]$ denotes $\hat{\tau}$ or ϵ. If the generated observation is not relevant, the reduction is written $t \rightarrow t'$.

Definition 1 (Observable Trace: $\omega \circ \mathcal{T}$). *Given an execution $e = t_0 \rightharpoonup \ldots (\rightharpoonup t_n)$, $\mathcal{T}(e)$ is the annotated reduction sequence $t'_0 \xrightarrow{[\hat{\tau}_1]} \ldots (\xrightarrow{[\hat{\tau}_n]} t'_n)$ where t'_i equals t_i with every constant annotated; in particular, every constant of t'_0 is annotated $c{:}c$. Furthermore $\omega(\mathcal{T}(e))$ denotes the* observable trace *of e, which is the sequence of observable actions $[\hat{\tau}_1], \ldots (, [\hat{\tau}_n])$ with the silent actions ϵ filtered out.*

Example 2. *Let the first execution in Example 1 be denoted by e. The annotated execution $\mathcal{T}(e)$ looks as follows:*

$$
\begin{aligned}
& \texttt{exec}\ (\lambda x.((\texttt{flipCoin} = \mathsf{hd}{:}\mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ \texttt{userInput}) \\
\xrightarrow{\mathsf{userInput}()}\ & \texttt{exec}\ (\lambda x.((\texttt{flipCoin} = \mathsf{hd}{:}\mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ \mathsf{userInput}(){:}s) \\
\xrightarrow{\epsilon}\ & \texttt{exec}\ ((\texttt{flipCoin} = \mathsf{hd}{:}\mathsf{hd})\ (\texttt{sanitize}\ \mathsf{userInput}(){:}s)\ \mathsf{userInput}(){:}s) \\
\xrightarrow{\mathsf{flipCoin}()}\ & \texttt{exec}\ ((\mathsf{flipCoin}(){:}\mathsf{hd} = \mathsf{hd}{:}\mathsf{hd})\ (\texttt{sanitize}\ \mathsf{userInput}(){:}s)\ \mathsf{userInput}(){:}s) \\
\xrightarrow{\epsilon}\ & \texttt{exec}\ (\texttt{sanitize}\ (\mathsf{userInput}(){:}s)) \\
\xrightarrow{\mathsf{sanitize}(\mathsf{userInput}())}\ & \texttt{exec}\ (\mathsf{sanitize}(\mathsf{userInput}()){:}s') \\
\xrightarrow{\mathsf{exec}(\mathsf{sanitize}(\mathsf{userInput}()))}\ & \texttt{exec}\ (\mathsf{sanitize}(\mathsf{userInput}())){:}\mathsf{r\_}s'
\end{aligned}
$$

The observable trace of e, $\omega(\mathcal{T}(e))$, is: $\mathsf{userInput}(), \mathsf{flipCoin}(), \mathsf{sanitize}(\mathsf{userInput}()), \mathsf{exec}(\mathsf{sanitize}(\mathsf{userInput}()))$.

The calculus ensures that the annotation of any constant fully captures how that value was computed, and filters out unrelated processing. In fact, if each constant c in a term t is annotated with c itself and $t \rightarrow^* \hat{\tau} : c'$ then $\hat{\tau}$ alone can be used to recover the computation resulting in the constant c'. This property is formalized in Theorem 1.

Theorem 1. *Given a term t in which all constants have the shape $c : c$, if $t \rightarrow^* \tau : c'$ then $term(\tau) \rightarrow^* \tau : c'$ where $term(f(\tau_1, \ldots, \tau_n)) = f^n\ term(\tau_1)\ \ldots\ term(\tau_n)$ and $term(c) = c{:}c$.*

Proof. *We start by showing that if, for all annotated constants $\tau : c$ in a term t, $term(\tau) \rightarrow^* \tau : c$ holds and $t \rightarrow t'$ then, for all annotated constants $\tau' : c'$ in t', $term(\tau') \rightarrow \tau' : c'$ holds . This is shown by induction on the derivation tree of $t \rightarrow t'$. All cases except* T-AppFun *are trivial as no other rule introduces a new constant. For the* T-AppFun *case we need to show that $term(f(\tau_1, \ldots, \tau_n)) \rightarrow^* term(f(\tau_1, \ldots, \tau_\mathsf{n}))\ :\ [\![f^n(c_1, \ldots, c_n)]\!]$. We first note that $term(f(\tau_1, \ldots, \tau_n)) = f^n\ term(\tau_1)\ \ldots\ term(\tau_n)$. Since we know that $term(\tau_i) \rightarrow^* \tau_i : c_i$ for all annotated constants in t, we have $f^n term(\tau_1) \ldots term(\tau_n) \rightarrow^* f^n\ \tau_1 : c_1\ \ldots\ \tau_\mathsf{n} : c_n \rightarrow f(\tau_1, \ldots, \tau_\mathsf{n}) : [\![f^n(c_1, \ldots, c_n)]\!]$.*

The result now follows from the fact that all constants in t are on the form $c{:}c$ and $term(c) \rightarrow^* c{:}c$. □

Note, however, that the trace of a reduction $t \rightarrow^* \tau{:}c$ may not be equal to the trace of $term(\tau) \rightarrow^* \tau{:}c$ since some computations may be discarded in the former reduction. In Example 2 for instance, `exec` (`sanitize` (`userInput`())) is sufficient to retrieve the core processing resulting in $\mathsf{r\_}s'$ (from which the `flipCoin` related code has been filtered out).

The choice of tracking direct function applications and not decisions regarding branching (i.e. tracking direct data flows and not indirect ones) is deliberate but not a fundamental requirement of the approach. The calculus could in principle be adapted to take branching decisions (explicit indirect flows) into account simply by (1) annotating terms with a context describing which computations the current computations depends upon and (2) update this context based on τ_1 and τ_2 in the T-CondT and T-CondF rules. However, from a theoretical point of view, the policies we currently have in mind are strongly related to data processing (i.e. *what* is actually computed rather than *under what conditions* something is computed) and can be conveniently enforced using existing taint tracking mechanisms (i.e. mechanisms tracking only direct flows). Moreover, from a practical point of view, the absence of efficient dynamic data tracking mechanisms for a commercial-level system handling indirect flows, on top of which to implement our approach, reinforces this choice.

3. POLICIES

In our setting, a policy $\mathcal{P}$ specifies which computations (nesting of function applications) are allowed to be performed. Since each new function application is recorded in the form of an observable action, policies can be conveniently expressed as a predicate over traces, i.e. sequences of observable actions. This nomenclature is standard in monitoring, [23, 1, 20]. A reduction sequence, e, is said to be *accepted* by $\mathcal{P}$ if and only if $\mathcal{P}(\omega(\mathcal{T}(e)))$ holds and *rejected*

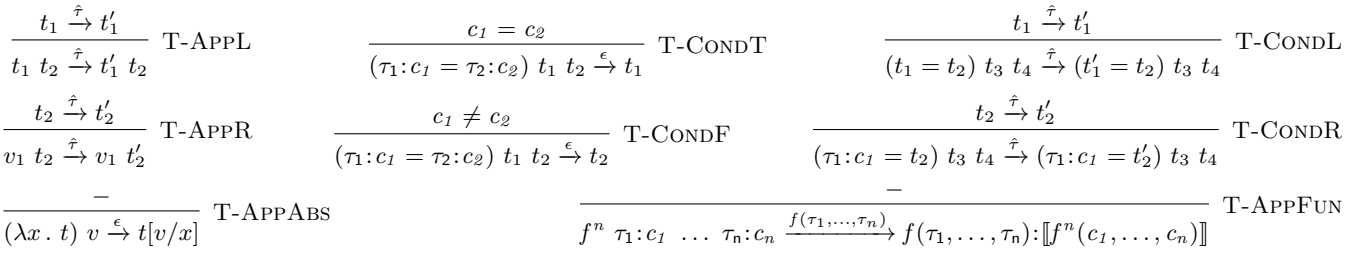

Figure 1: Reduction rules with history annotations and observable actions.

otherwise. In this paper, we do not consider arbitrary policies. Some policies can not even be enforced in practice. The class of policies considered include only the ones that are *local* and *subtree closed*.

Definition 2 (Local Policy). *A policy, $\mathcal{P}$, is said to be* local *if a predicate P exists such that $P(c)$ holds for all $c \in \mathsf{C}$ and $\mathcal{P}(\hat{\tau}_1, \ldots (, \hat{\tau}_n))$ holds iff $\forall_{0 \leq i\ (<n)}\ P(\hat{\tau}_i)$ holds.*

This property allows us to focus on stateless policies stating *which* computations may be performed rather than *when* they may be performed and alleviates the need of a global monitor state. This constraint is however not fundamental and can be relaxed by instead stating that the set of accepted traces should be prefix-closed (i.e. that if a trace is accepted then so should all its prefixes). This would allow policies to express temporal properties of the observable traces, such as *"f may not be evaluated until g has been evaluated"* and would arguably be more suitable when, for instance, dealing with functions with side-effects. Such class of policies has however been studied in depth already, [7, 8, 1] and we see no incompatibility with those studies and the results in this paper.

Definition 3 (Subtree Closed Policy). *A local policy is* subtree closed *if the set of observable actions for which P (Definition 2) holds is subtree closed.*

This property rules out policies that for instance accept the evaluation of $g(f())$ but rejects the evaluation of $f()$. As discussed below such policies are not meaningful in languages with call-by-value semantics like Java, since $f()$ indeed needs to be evaluated in order to evaluate $g(f())$ (in a call-by-name setting however, $f()$ does not necessarily need to be evaluated).

Example 3. *A typical example of a policy which is local and subtree closed could for instance express that* `sanitize` *accepts any string, while* `exec` *only accepts results from the* `sanitize` *function (i.e. all strings must be sanitized before they are passed to* `exec`*). With such a policy, the evaluation of the term of Example 1 is accepted if* `flipCoin` *returns* `hd` *and rejected otherwise.*

The hierarchy between prefix-closed, local and subtree-closed policies, both in general and under the assumption of a call-by-value semantics (CBV), can be summarized as follows: subtree-closed policies are by definition also local; local policies are not necessarily subtree closed except for CBV semantics; local policies are prefix-closed but prefix-closed policies are not necessarily local, not even when assuming a CBV semantics. The distinction between call-by-value and call-by-name semantics come from the fact that, in CBV, any parameter of an executed function call has been evaluated previously.

Lemma 1. *In CBV semantics, if $\hat{\tau}_1, \ldots, \hat{\tau}_n, f^i(\tau_j, \ldots, \tau_k)$ is a prefix of a trace of an execution $\omega(\mathcal{T}(t_0 \rightharpoonup \ldots (\rightharpoonup t_m)))$, then $(\{\tau_j, \ldots, \tau_k\} \setminus \mathsf{C}) \subseteq \{\hat{\tau}_1, \ldots, \hat{\tau}_n\}$.*

Proof. *Any argument of f^i must have the form $\tau_l : c$. Either c comes from the original term, in which case $\tau_j \in \mathsf{C}$ or c is the result of a function application prior to $f^i(\tau_j, \ldots, \tau_k)$ in which case $\tau_j \in \{\hat{\tau}_1, \ldots, \hat{\tau}_n\}$.* □

4. LABELS

Manipulating FATs at runtime for enforcement is not efficient in practice. For instance, if a policy requires an argument to be the result of an even number of applications of `toggle`, the FATs could grow indefinitely, despite the fact that a boolean value would suffice to maintain and describe the relevant computational history. To circumvent this problem, *labels* are introduced to replace FATs. A label can be seen as an abstraction of a FAT.

Labels (denoted by α, β, ...) range over a set L of ground labels closed under $\oplus$. A special label $\mathsf{L}_0 \in \mathsf{L}$ denotes the default label and is used for constants present in the initial term. By convention, L_f denotes the label associated to f^n (L_f can be the same as L_g). The deterministic $\oplus$ operator is used to compute the label of the result of a function application. $\oplus$ can be seen as an abstraction of the FAT constructor. The pair $\langle \mathsf{L}, \oplus \rangle$ is referred to as a *labeling scheme*. It has the nice property to be instantiable to reflect the taint tracking policy of TaintDroid, on which our implementation is based. For enforcement purposes, programs are evaluated using a new semantics $\rightarrow_\oplus$ manipulating labels similar to the one of Figure 1 where observable actions are removed and tree-annotated constants $\tau_i : c_i$ are replaced by label-annotated constants $\alpha_i : c_i$. The only difference is the rule T-AppFun which is replaced by the rule L-AppFun which follows:

$$\frac{\gamma = \mathsf{L}_f \oplus \alpha_1 \oplus \cdots \oplus \alpha_n}{f^n\ \alpha_1 : c_1\ \ldots\ \alpha_n : c_n \rightarrow_\oplus \gamma : [\![f^n(c_1, \ldots, c_n)]\!]}$$

Just as in the case with FATs, the labels do not affect the resulting terms and every annotated reduction sequence has a corresponding unannotated reduction sequence. *strip*(t) is t with the label of every constant removed and *init*(t) is the term t in which each unlabeled constant c is replaced by $\mathsf{L}_0 : c$.

Proposition 1. *If $t \rightarrow_\oplus^n t'$ then strip$(t) \rightharpoonup^n$ strip(t').*

The reverse is true for every term only if $\oplus$ is a total function. It is a potential property of an execution.

DEFINITION 4 (VALID LABELING). *An execution* $t_0 \rightharpoonup^n t_n$ *has a* valid $\langle \mathsf{L}, \oplus \rangle$-labeling *iff there exists* $t'_1, \ldots, t'_n$ *such that* $init(t_0) \rightarrow^n_{\oplus} t'_n$ *where* $t_i = strip(t'_i)$ *for* $i \in [1, n]$.

By allowing $\oplus$ to be a partially defined function certain sequences of reductions are ruled out due to the premise of the L-APPFUN rule. This can be (and is) exploited as an enforcement mechanism as shown in the following definition.

DEFINITION 5 ($\langle \mathsf{L}, \oplus \rangle$ ENFORCEMENT). $\langle \mathsf{L}, \oplus \rangle$ enforces *a policy* $\mathcal{P}$ *if all executions with valid* $\langle \mathsf{L}, \oplus \rangle$*-labelings are accepted by* $\mathcal{P}$*. If the reverse also holds, i.e. all executions accepted by* $\mathcal{P}$ *have a valid* $\langle \mathsf{L}, \oplus \rangle$*-labeling,* $\langle \mathsf{L}, \oplus \rangle$ *is said to* precisely enforce $\mathcal{P}$.

An example of how a labeling scheme enforces a *sanitize before executing*-policy is presented in Example 5 in Section 6.

When reasoning about the correctness of the labeling scheme we need a way to tell which label a certain FAT corresponds to. For this purpose we define the $R_{\mathsf{L},\oplus}$-function as follows:

DEFINITION 6 ($R_{\mathsf{L},\oplus}$). $R_{\mathsf{L},\oplus}(\tau)$ *is defined as follows:*

$$R_{\mathsf{L},\oplus}(c) = \mathsf{L}_0$$
$$R_{\mathsf{L},\oplus}(f(\tau_1, \ldots, \tau_n)) = \mathsf{L}_f \oplus R_{\mathsf{L},\oplus}(\tau_1) \oplus \ldots \oplus R_{\mathsf{L},\oplus}(\tau_n)$$

LEMMA 2 (LABELS ABSTRACT FATS). *For any* t_0*:*

$$t_o \rightarrow^n \tau{:}c \ \wedge \ R_{\mathsf{L},\oplus}(\tau) \in \mathsf{L} \ \Leftrightarrow \ init(t_0) \rightarrow^n_{\oplus} \mathsf{R}_{\mathsf{L},\oplus}(\tau){:}c$$

PROOF. *By induction on the length of the derivation. The only interesting case,* T-APPFUN*, follows directly from the semantics definitions and Definition 6.* □

An execution is accepted iff, for any observable action $\hat{\tau}$ generated, $R_{\mathsf{L},\oplus}(\hat{\tau})$ is defined.

THEOREM 2 (ACCEPTED EXECUTION). *Execution* $e = t_0 \rightharpoonup^n t_n$ *has a valid* $\langle \mathsf{L}, \oplus \rangle$*-labeling* ($init(t_0) \rightarrow^n_{\oplus} t'_n$) *iff, for any* $\hat{\tau}$ *in* $\omega(\mathcal{T}(e))$*,* $R_{\mathsf{L},\oplus}(\hat{\tau})$ *is defined* ($R_{\mathsf{L},\oplus}(\hat{\tau}) \in \mathsf{L}$).

PROOF. *Assuming* $\rightarrow$ *goes through,* $\rightarrow_{\oplus}$ *can only fail on* L-APPFUN*, whose corresponding rule* T-APPFUN *is the only one generating observable FATs. Hence, if* $R_{\mathsf{L},\oplus}(\hat{\tau})$ *is defined for any* $\hat{\tau}$ *in* $\omega(\mathcal{T}(e))$ *then Lemma 2 helps conclude that* $init(t_0) \rightarrow^n_{\oplus} t'_n$*. The other direction is proved by induction on the length of* $init(t_0) \rightarrow^n_{\oplus} t'_n$ *and by observing that, by Lemma 1 and induction hypothesis, for any subtree* τ *of the potentially newly generated FAT* $\hat{\tau}_{n+1}$*,* $R_{\mathsf{L},\oplus}(\tau)$ *is defined and, by Lemma 2, equal to the corresponding label in* t'_n*. Hence,* $R_{\mathsf{L},\oplus}(\hat{\tau}_{n+1})$ *is defined.* □

5. DEFINING AND ENFORCING POLICIES

As suggested previously, security policies in this work are defined by a bottom-up *deterministic finite tree automaton* (DFTA) [5] that characterizes a set of FATs.

DEFINITION 7 (DFTA [5]). *A (bottom-up)* deterministic finite tree automaton *over a ranked alphabet* A *is a tuple* $\mathcal{A} = (Q, A, Q_f, \Delta)$ *where* Q *is a set of (unary) states,* $Q_f \subseteq Q$ *is a set of accepting states and* Δ *is a partial function defined by a set of transition rules of the form* $a(q_1(\mathsf{t}_1), \ldots, q_n(\mathsf{t}_n)) \rightarrow q(a(\mathsf{t}_1, \ldots, \mathsf{t}_n))$ *where* $n \geq 0$*,* $a \in A_n$*,* $q_1, \ldots, q_n \in Q$ *and* $\mathsf{t}_1, \ldots, \mathsf{t}_n$ *are terms over* A*.*

A ground term t *of* A *is accepted by an automata* $\mathcal{A}$ ($\mathsf{t} \in \mathcal{L}(\mathcal{A})$) *if* $\mathsf{t} \rightarrow^* q(\mathsf{t})$ *for some* $q \in Q_f$*.*

Intuitively, an automaton accepts a term t iff every node in the tree t can be annotated with a state such that the root node is annotated with a final state, and the annotations are compatible with the transition rules Δ.

A policy is defined in terms of a DFTA over $\mathsf{F} \cup \mathsf{C}$, from now on referred to as a *security tree automaton* (STA).

DEFINITION 8 (STA). *A* security tree automaton *is a DFTA* $\mathcal{A} = (Q, \mathsf{F} \cup \mathsf{C}, Q, \Delta)$ *such that there exists* q_0 *in* Q *and for all* c *in* C*:* $c \rightarrow q_0(c) \in \Delta$.

$\mathcal{P}_{\mathcal{A}}$ denotes the policy defined by the STA $\mathcal{A}$ whose set of accepted traces is $\{[\hat{\tau}_0, \ldots (, \hat{\tau}_n)] \mid \forall i \in [0, n].\ \hat{\tau}_i \in \mathcal{L}(\mathcal{A})\}$.

LEMMA 3. *A policy defined in terms of a STA is local.*

PROOF. *The predicate* $P(\tau) \stackrel{\text{def}}{=} \tau \in (\mathsf{C} \cup \mathcal{L}(\mathcal{A}))$ *is a valid candidate showing that the policy* $\mathcal{P}_{\mathcal{A}}$ *is local according to Definition 2.* □

Additionally, as for ordinary security automata [29], all states of a STA are required to be accepting (this does not imply that all trees are accepted, since Δ is partial). This requirement is sufficient to ensure that STA policies are subtree closed.

LEMMA 4. *The language of a STA is subtree closed.*

PROOF. *If* $f(\tau_1, \ldots, \tau_n)$ *is accepted, then there exist* q*,* $q_1, \ldots, q_n$ *such that* $f(\tau_1, \ldots, \tau_n) \rightarrow^* f(q_1(\tau_1), \ldots, q_n(\tau_n)) \rightarrow q(f(\tau_1, \ldots, \tau_n))$*. This means that, for all* τ_i *in* $\tau_1, \ldots, \tau_n$*,* $\tau_i \rightarrow^* q_i(\tau_i)$ *and since* $Q_f = Q$*,* τ_i *is also accepted.* □

If there exists an injective function from the states of an STA $\mathcal{A}$ to the labels of a labeling scheme $\langle \mathsf{L}, \oplus \rangle$ and $\oplus$ simulates the transitions in Δ then $\langle \mathsf{L}, \oplus \rangle$ precisely enforces $\mathcal{P}_{\mathcal{A}}$.

THEOREM 3 (EQUIVALENT $\langle \mathsf{L}, \oplus \rangle$). $\langle \mathsf{L}, \oplus \rangle$ *precisely enforces the policy described by the STA* $\mathcal{A} = (Q, \mathsf{F} \cup \mathsf{C}, Q, \Delta)$ *if there exists an injective function* $L : Q \rightarrow \mathsf{L}$ *such that, with* $\mathsf{L}_c = \mathsf{L}_0$ *for all* c *in* C*:*

$$L(q) = \mathsf{L}_a \oplus L(q_1) \oplus \ldots \oplus L(q_n) \iff a(q_1(\tau_1), \ldots, q_n(\tau_n)) \rightarrow q(a(\tau_1, \ldots, \tau_n)) \in \Delta \quad (1)$$

PROOF. *Following Definition 5, it is sufficient that for a given* L *the following holds for any execution* $e = t_0 \rightharpoonup^n t_n$*:*

$$e \text{ has a valid } \langle \mathsf{L}, \oplus \rangle\text{-labeling} \iff e \text{ is accepted by } \mathcal{P}_{\mathcal{A}}$$

Since the policy is local and since all states in the policy automaton are accepting, by Lemma 2 and Definition 8, it is sufficient to show the following for each $\hat{\tau} \in \omega(\mathcal{T}(e))$*:*

$$R_{\mathsf{L},\oplus}(\hat{\tau}) \in \mathsf{L} \iff \exists q.\ \hat{\tau} \rightarrow^* q(\hat{\tau})$$

which follows from the following holding for all τ*:*

$$\tau \rightarrow^* q(\tau) \ \Rightarrow \ R_{\mathsf{L},\oplus}(\tau) = L(q) \quad (2)$$
$$R_{\mathsf{L},\oplus}(\tau) = \alpha \ \Rightarrow \ \tau \rightarrow^* L^{-1}(\alpha)(\tau) \quad (3)$$

(2) and (3) follow by induction on τ *by assuming (1).* □

We now turn to the syntax of the language for describing security tree automata. We first give a concrete self explanatory example and then generalize this into a proper definition.

EXAMPLE 4. *A policy stating that the function* `exec` *only accepts strings returned by another function* `sanitize`*, or concatenations of any such strings is written as:*

$$\begin{array}{ll}\{\mathsf{unsanitized}, \mathsf{sanitized}\},\\ \{\mathtt{sanitize}(\alpha): & \mathit{true} \to \mathsf{sanitized}\\ \mathtt{concat}(\alpha_1, \alpha_2): & \alpha_1 = \alpha_2 = \mathsf{sanitized} \to \mathsf{sanitized}\\ & \mathit{true} \to \mathsf{unsanitized}\\ \mathtt{exec}(\alpha): & \alpha = \mathsf{sanitized} \to \mathsf{sanitized}\}\end{array}$$

Where the default label L_0 *equals* unsanitized*.*

The general syntax of a policy is defined as follows.

DEFINITION 9 (POLICY SYNTAX). *Syntactically, a policy is expressed as follows*

$$\begin{array}{ll}\{\mathsf{L}_0, \mathsf{L}_1, \ldots, \mathsf{L}_{n_L}\},\\ \{f_1^{n_1}(\alpha_{11}, \ldots, \alpha_{1n_1}): & \mathit{guard}_{11} \to \mathit{expr}_{11}\\ & \ldots\\ & \mathit{guard}_{1g_1} \to \mathit{expr}_{1g_1},\\ \vdots & \\ f_k^{n_k}(\alpha_{k1}, \ldots, \alpha_{kn_k}): & \mathit{guard}_{k1} \to \mathit{expr}_{k1}\\ & \ldots\\ & \mathit{guard}_{kg_k} \to \mathit{expr}_{kg_k}\}\end{array}$$

where each guard is a boolean expression and each expr is a label expression, both of which are composed from the declared label constants, the argument labels and simple tests such as equality.

Intuitively, when $f_i^{n_i}$ is invoked the formal parameters, $\alpha_{i1}, \ldots, \alpha_{in_i}$, are bound to the labels associated with the arguments. The return label is computed from the expression expr_{ij} corresponding to the first guard guard_{ij} that holds among $\mathit{guard}_{i1}, \ldots, \mathit{guard}_{ig_i}$. If no guard holds, the invocation is to be seen as a violation of the policy. A formal translation into a security tree automata follows.

DEFINITION 10 (POLICY SEMANTICS). *Given a policy* $\mathcal{P}$ *in the syntax of Definition 9, the corresponding security tree automaton,* $\mathcal{A}_\mathcal{P} = (Q, \mathsf{F} \cup \mathsf{C}, Q, \Delta)$*, is defined as follows:* $Q = \{q_0, q_1, \ldots, q_{n_L}\}$ *and* $\Delta = \{c \twoheadrightarrow q_0(c) \mid c \in \mathsf{C}\} \cup \bigcup \delta_{ij}$*, where each* δ_{ij} *represents the set of automaton transitions corresponding to guard* j *in clause* i*:*

$$\begin{array}{l}\{\ f_i(q_1'(x_1), \ldots, q_{n_i}'(x_{n_i})) \twoheadrightarrow q'(f_i(x_1, \ldots, x_{n_i}))\\ \mid\ q_1' \ldots q_{n_i}' \in Q\ \wedge\ q' = [\![\mathit{expr}_{ij}[q_k/\mathsf{L}_k][q_k'/\alpha_{ik}]]\!]\\ \quad \wedge\ [\![\mathit{guard}_{ij}[q_k/\mathsf{L}_k][q_k'/\alpha_{ik}]]\!]\\ \quad \wedge \neg \bigvee_{1 \le g < j} [\![\mathit{guard}_{ig}[q_k/\mathsf{L}_k][q_k'/\alpha_{ik}]]\!]\ \}\end{array}$$

As mentioned in Section 4, the enforcement mechanism does not, for practical reasons, work directly on FATs but on labels. To enforce a policy, an equivalent labeling scheme is needed. For policies expressed in the syntax of Definition 9, such labeling scheme can be defined as follows.

DEFINITION 11. *Given a policy* $\mathcal{P}$ *in the syntax of Definition 9, the labeling scheme* $LS(\mathcal{P})$ *is defined as* $\langle\{\mathsf{L}_0, \mathsf{L}_1, \ldots, \mathsf{L}_{n_L}\} \cup \{\mathsf{L}_{f_i}\}, \oplus\rangle$ *where* $\mathsf{L}_{f_i} \oplus \alpha_1 \oplus \cdots \oplus \alpha_{n_i}$ *is:*

$$\begin{array}{ll}\mathit{if}\ [\![\mathit{guard}_{i1}[\alpha_k/\alpha_{ik}]]\!] & \mathit{then}\ [\![\mathit{expr}_{i1}[\alpha_k/\alpha_{ik}]]\!]\\ \mathit{else\ if} \ldots & \mathit{then} \ldots\\ \mathit{else\ if}\ [\![\mathit{guard}_{ig_i}[\alpha_k/\alpha_{ik}]]\!] & \mathit{then}\ [\![\mathit{expr}_{ig_i}[\alpha_k/\alpha_{ik}]]\!]\end{array}$$

We now show that $LS(\mathcal{P})$ indeed precisely enforces $\mathcal{P}$ according the semantics defined in Definition 10.

THEOREM 4 (CORRECTNESS OF LS). *Given a policy* $\mathcal{P}$ *in the syntax of Definition 9,* $LS(\mathcal{P})$ *enforces* $\mathcal{P}$ *precisely.*

PROOF. *Follows directly from Theorem 3 with* $L(q_i) = \mathsf{L}_i$ □

6. LABELED IMPERATIVE LANGUAGE

A λ-calculus is a natural candidate for the core language since central concepts such as function applications are easily represented. Furthermore, the structure and potential data flow in a program is arguably clearer if written in the form of a λ-term than in a language with side effects. Nonetheless it is important to make sure that the calculus is general enough to serve as a foundation for other languages as well, such as Java which is the language of applications monitored by TreeDroid. This section introduces a While-language whose semantics track labels in a natural way. A straightforward encoding of the language in our calculus is then provided and shown to agree with the labeling semantics.

The language presented in this section is a simple imperative language with loops and the addition of labels, external function applications and return statement. The small-step operational semantics of the language is given in Figure 2. A configuration is represented as $\langle S, \sigma\rangle$ where S is the statement to be executed and σ the current store. The store maps variables to labeled values and the initial store, σ_0, maps each variable to L_0:0.

EXAMPLE 5. *The program in this exmaple is similar to the one of Example 1.*

```
x1 := userInput();
if flipCoin() = 1 then x1 := sanitize(x1) else x1 := x1;
return exec(x1)
```

With the labeling scheme, $\langle\{\mathsf{L}_0, \mathsf{input}, \mathsf{sanitized}, \mathsf{result}, \mathsf{L}_{exec}, \mathsf{L}_{flipCoin}, \mathsf{L}_{sanitize}, \mathsf{L}_{userInput}\}, \oplus\rangle$ *where*

$$\alpha_1 \oplus \alpha_2 \oplus \ldots \oplus \alpha_m = \begin{cases}\mathsf{L}_0 & \mathit{if}\ \alpha_1 = \mathsf{L}_{flipCoin}\\ \mathsf{input} & \mathit{if}\ \alpha_1 = \mathsf{L}_{userInput}\\ \mathsf{sanitized} & \mathit{if}\ \alpha_1 = \mathsf{L}_{sanitize}\\ \mathsf{result} & \mathit{if}\ \begin{cases}\alpha_1 = \mathsf{L}_{exec}\\ \alpha_2 = \mathsf{sanitized}\end{cases}\end{cases}$$

an execution in which `flipCoin` *yields 1 terminates:*

$$\begin{array}{ll} & \langle \mathtt{x}_1 := \mathtt{userInput}();\ \mathtt{if}\ \mathtt{flipCoin}() = 1\ \mathtt{then}\ \ldots,\ \sigma_0\rangle\\ \twoheadrightarrow_\oplus & \langle \mathtt{if}\ \mathtt{flipCoin}() = 1\ \mathtt{then}\ \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle\\ \twoheadrightarrow_\oplus & \langle \mathtt{x}_1 := \mathtt{sanitize}(\mathtt{x}_1); \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle\\ \twoheadrightarrow_\oplus & \langle \mathtt{return}\ \mathtt{exec}(\mathtt{x}_1),\ \sigma_0[x_1 \mapsto \mathsf{sanitized{:}7'}]\rangle\\ \twoheadrightarrow_\oplus & \mathsf{result{:}7''}\end{array}$$

and an execution in which `flipCoin` *yields 0 would get stuck:*

$$\begin{array}{ll} & \langle \mathtt{x}_1 := \mathtt{userInput}();\ \mathtt{if}\ \mathtt{flipCoin}() = 1\ \mathtt{then}\ \ldots,\ \sigma_0\rangle\\ \twoheadrightarrow_\oplus & \langle \mathtt{if}\ \mathtt{flipCoin}() = 1\ \mathtt{then}\ \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle\\ \twoheadrightarrow_\oplus & \langle \mathtt{x}_1 := \mathtt{x}_1; \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle\\ \twoheadrightarrow_\oplus & \langle \mathtt{return}\ \mathtt{exec}(\mathtt{x}_1),\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle\end{array}$$

since $\mathsf{L}_{exec} \oplus \mathsf{input}$ *is undefined.*

Figure 3 gives an encoding of While in the style of state transformers with $\mathsf{C} = \mathbb{N}$ and $\mathsf{F} = \{f_1, \ldots, f_k\}$.

$$\frac{-}{\langle n,\sigma\rangle \twoheadrightarrow_{\oplus} \mathsf{L}_0 : n} \qquad \frac{-}{\langle \mathrm{x}_k,\sigma\rangle \twoheadrightarrow_{\oplus} \sigma(\mathrm{x}_k)} \qquad \frac{\langle E_1,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n_1 \quad \ldots \quad \langle E_m,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_m : n_m}{\langle f(E_1,\ldots,E_m),\sigma\rangle \twoheadrightarrow_{\oplus} \mathsf{L}_f \oplus \alpha_1 \oplus \cdots \oplus \alpha_m : [\![f(n_1,\ldots,n_m)]\!]}$$

$$\frac{\langle E,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha : n}{\langle \mathrm{x}_k := E,\sigma\rangle \twoheadrightarrow_{\oplus} \sigma[\mathrm{x}_k \mapsto \alpha : n]} \qquad \frac{\langle S_1,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S_1',\sigma'\rangle}{\langle S_1;S_2,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S_1';S_2,\sigma'\rangle} \qquad \frac{\langle S_1,\sigma\rangle \twoheadrightarrow_{\oplus} \sigma'}{\langle S_1;S_2,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S_2,\sigma'\rangle} \qquad \frac{\langle E,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha : n}{\langle \textbf{return } E,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha : n}$$

$$\frac{\langle E_1,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n \quad \langle E_2,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n}{\langle \textbf{if } E_1 = E_2 \textbf{ then } S_1 \textbf{ else } S_2,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S_1,\sigma\rangle} \qquad \frac{\langle E_1,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n_1 \quad \langle E_2,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n_2 \quad n_1 \neq n_2}{\langle \textbf{if } E_1 = E_2 \textbf{ then } S_1 \textbf{ else } S_2,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S_2,\sigma\rangle}$$

$$\frac{\langle E_1,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n \quad \langle E_2,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n}{\langle \textbf{while } E_1 = E_2 \textbf{ do } S,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S;\textbf{while } E_1 = E_2 \textbf{ do } S,\sigma\rangle} \qquad \frac{\langle E_1,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n_1 \quad \langle E_2,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n_2 \quad n_1 \neq n_2}{\langle \textbf{while } E_1 = E_2 \textbf{ do } S,\sigma\rangle \twoheadrightarrow_{\oplus} \sigma}$$

Figure 2: Semantics of the While-language with Function Application Monitoring.

Auxiliary definitions:

$$\begin{aligned} \mathit{get} &= \lambda s.\,\lambda x.\,s\ x \\ \mathit{set} &= \lambda s.\,\lambda x.\,\lambda v.\,\lambda k.\,(k = x)\ v\ (s\ k) \\ \mathit{fix} &= \lambda g.\,(\lambda x.\,g\ (\lambda y.\,x\ x\ y))\ (\lambda x.\,g\ (\lambda y.\,x\ x\ y)) \end{aligned}$$

Expressions:

$$\begin{aligned} [\![n]\!] &= \lambda s.\,\mathsf{L}_0 : n \\ [\![\mathrm{x}_k]\!] &= \lambda s.\,\mathit{get}\ s\ k \\ [\![f(E_1,\ldots,E_m)]\!] &= \lambda s.\,f\ ([\![E_1]\!]\ s)\ \ldots\ ([\![E_m]\!]\ s) \end{aligned}$$

Statements:

$$\begin{aligned} [\![S_1;S_2]\!] &= \lambda s.\,[\![S_2]\!]\ ([\![S_1]\!]\ s) \\ [\![\mathrm{x}_k := E]\!] &= \lambda s.\,\mathit{set}\ s\ k\ ([\![E]\!]\ s) \\ [\![\textbf{if } E_1 = E_2 \textbf{ then } S_1 \textbf{ else } S_2]\!] &= \lambda s.\,([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ ([\![S_1]\!]\ s)\ ([\![S_2]\!]\ s) \\ [\![\textbf{while } E_1 = E_2 \textbf{ do } S]\!] &= \mathit{fix}\ (\lambda w.\,\lambda s.\,([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s) \\ [\![\textbf{return } E]\!] &= \lambda s.\,[\![E]\!]\ s \end{aligned}$$

Figure 3: Encoding of While in our λ-calculus

Definition 12. *A state is a term $\lambda k.\,t$ which reduces to a labeled constant when applied to a number: $(\lambda k.\,t)\ n \rightarrow^* \alpha{:}c$. A state s and a store σ* agree, *$s \sim \sigma$, iff $s\ k \rightarrow^* \sigma(\mathrm{x}_k)$ for all k.*

Initial state for evaluation is $s_0 = \lambda k.\,\mathsf{L}_0 : 0$. The following lemmas show that $\twoheadrightarrow_{\oplus}$ evaluating P accepts the same executions as $\rightarrow_{\oplus}$ evaluating $[\![\mathrm{P}]\!]$.

Lemma 5 (Expression equivalence). *If $s \sim \sigma$ then $\langle E,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha{:}n \Leftrightarrow [\![E]\!]\ s \rightarrow^*_{\oplus} \alpha{:}n$.*

Proof. *By structural induction on E. For $E \equiv n$, we have $\langle n,\sigma\rangle \twoheadrightarrow_{\oplus} \mathsf{L}_0{:}n$ and $[\![n]\!]\ s \rightarrow_{\oplus} \mathsf{L}_0{:}n$. For $E \equiv x_k$, we have $\langle \mathrm{x}_k,\sigma\rangle \twoheadrightarrow_{\oplus} \sigma(\mathrm{x}_k)$ and $[\![\mathrm{x}_k]\!]\ s \rightarrow_{\oplus} (\lambda s.\mathit{get}\ s\ k)\ s \rightarrow_{\oplus} \mathit{get}\ s\ k \rightarrow_{\oplus} s\ k$ where $s\ k$ reduces to $\sigma(\mathrm{x}_k)$ by Definition 12. For $E \equiv f(E_1,\ldots,E_m)$, $\langle f(E_1,\ldots,E_m),\sigma\rangle \twoheadrightarrow_{\oplus} \mathsf{L}_f \oplus \alpha_1 \oplus \ldots \oplus \alpha_m : [\![f(n_1,\ldots,n_m)]\!]$, assuming $\langle E_i,\sigma\rangle \twoheadrightarrow_{\oplus} \alpha_i{:}n_i$. By induction, $[\![f(E_1,\ldots,E_m]\!]\ s = \lambda s.f\ ([\![E_1]\!]s)\ \ldots\ ([\![E_m]\!]\ s)$ reduces to $f\ \alpha_1{:}n_1\ \ldots\ \alpha_m{:}n_m$ and then $\mathsf{L}_f \oplus \alpha_1 \oplus \ldots \oplus \alpha_m : [\![f(n_1,\ldots,n_m)]\!]$.* □

Lemma 6 (Statement equivalence). *If $s \sim \sigma$ then $\langle S,\sigma\rangle \twoheadrightarrow_{\oplus} \langle S',\sigma'\rangle$ (resp. $\alpha : n$) iff $[\![S]\!]\ s \rightarrow^*_{\oplus} [\![S']\!]\ s'$ (resp. $\alpha{:}n$) and $\sigma' \sim s'$ (assuming reductions proceed).*

Proof Sketch. *For brevity, full details and sublemmas are left out. In full details, the proof proceeds by defining $\rightarrow^+_{\oplus} \subseteq \rightarrow^*_{\oplus}$ step-wise equivalent to $\twoheadrightarrow_{\oplus}$. Briefly, the direction from $\twoheadrightarrow_{\oplus}$ to $\rightarrow^*_{\oplus}$ proceeds by cases on the semantics rules, and by observing that P evaluates to a value only if it evaluates a* **return** *and then $[\![P]\!]$ also evaluates to a value. The most complex cases are for the first* **while***-rule:*

$$\begin{aligned} & [\![\textbf{while } E_1 = E_2 \textbf{ do } S]\!]\ s \\ =\ & (\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ s \\ \rightarrow_{\oplus}\ & (\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ ((\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s))\ s)\ s \\ \rightarrow_{\oplus}\ & ([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ ((\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s))\ s \\ \rightarrow^*_{\oplus}\ & (\alpha_1 : n = \alpha_2 : n)\ ((\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s))\ s \\ \rightarrow_{\oplus}\ & (\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s) \\ =\ & [\![S;\textbf{while } E_1 = E_2 \textbf{ do } S]\!]\ s \end{aligned}$$

and for assignments where, by Lemma 5, $\langle \mathrm{x}_k := E,\sigma\rangle$ reduces to $\sigma[\mathrm{x}_k \mapsto \alpha{:}n]$ iff $[\![\mathrm{x}_k := E]\!]\ s = (\lambda s.\mathit{set}\ s\ k\ ([\![E]\!]\ s))\ s$ reduces to $\lambda l.(l = k)\ \alpha : n\ (s\ k)$ and, since $\sigma \sim s$, $\sigma[\mathrm{x}_k \mapsto \alpha{:}n] \sim \lambda l.(l = k)\ \alpha : n\ (s\ k)$. The other direction proceeds similarly by cases on $\rightarrow^+_{\oplus}$. □

7. IMPLEMENTATION

From a formal point of view, there is a large gap between the While-language and a real world language such as Java. Conceptually however, the semantics presented in the previous section outlines how the implementation of the labeling semantics works for Java. This section describes an implementation targeting Java (bytecode) and the Android platform which follows this outline. Implementing the framework involves solving two main tasks: tracking data flows and intercepting policy relevant function calls. In our implementation we solve the first task using *taint analysis* and the second task using *monitor inlining*.

7.1 Tracking Data Flows using Taint Analysis

Taint analysis (also known as taint tracking) is a common technique for tracking data flows at runtime. The technique relies on (1) having points at which data is originally tainted (*taint sources*), (2) making sure that taints propagate along with every data flow (*taint propagation*) and (3) having points at which taints of output data is intercepted (*taint sinks*). In our work we rely on taint propagation for tracking data flows by letting taints represent labels. The notion of taint sources and sinks however are factored out and handled by the inlined monitor.

Taint analysis implementations targeting the JVM has been described by several authors [18, 34, 4, 10]. Our implementation uses the TaintDroid framework by Enck et al [10] which targets the Android Platform. TaintDroid is based on a modified version of the Dalvik VM which taints data coming from various privacy related sources such as the GPS,

camera, microphone etc. and monitors the taints of data being sent on the network.

7.1.1 Limitations due to Taint Analysis

In TaintDroid a taint is represented by a 32-bit word where each bit corresponds to one of the privacy related sources. If for instance the taint of a value v has bit 25 and bit 32 is set, then v contains data which potentially comes from the camera and GPS respectively. When data is copied from one location to another, the taint is copied along with it. If two values v_1 and v_2 are combined (added or concatenated for instance) the taint of the result is determined by the bitwise or of the taint of v_1 and the taint of v_2. While this approach makes sense when working with the type of privacy related policies which TaintDroid is intended for, it poses a limitation on what policies we are able to enforce in our framework. For a policy to be enforceable, when using TaintDroid as the underlying taint tracking mechanism, it must have the two properties described below.

PROPERTY 1. *If L is the set of labels in a policy $\mathcal{P}$, there need to exist an injective function F of type $\mathsf{L} \to 2^{\{1,\ldots,32\}}$ such that the range of F is closed under $\cup$.*

This property ensures that for any two labels L_1 and L_2 there exists a label L_3 such that $F(\mathsf{L}_1) \cup F(\mathsf{L}_2) = F(\mathsf{L}_3)$ or, put differently, the bitwise or-operation performed by TaintDroid always yields a taint representing a valid label.

The other property is regarding arithmetic operations. Such operations are encoded as external functions in the theory, but in practice do not correspond to observable actions.

PROPERTY 2. *Whenever an arithmetic operation is applied to two labeled constants $\mathsf{L}_1 : c_1$ and $\mathsf{L}_2 : c_2$ the policy must define the resulting label as $F^{-1}(F(\mathsf{L}_1) \cup F(\mathsf{L}_2))$ where F is the function described in Property 1.*

In terms of abstract algebra, Property 1 and 2 hold for a policy $\mathcal{P}$ iff there exists a monomorphism between the two algebras $(\mathsf{L}, \odot)$ and $(2^{\{1,\ldots,32\}}, \cup)$ where $\odot$ is the label operator for arithmetic operations induced by $\mathcal{P}$. Our implementation assumes that these properties hold for all policies given to the inliner, and does not have syntactical support for defining custom behavior for arithmetic operations.

7.1.2 Taint Propagation: Or vs And

Using bitwise *or* as taint propagation mechanism is suitable when handling confidentiality properties. When for example enforcing a policy such as *"do not send address book data on the network"*, a phone number should *maintain* its taint, even if it is manipulated. Conceptually however, our framework works just as well for integrity related properties, such as *"send text messages only to numbers from my address book"*. Enforcing such policies call for a bitwise *and* propagation mechanism since if a phone number is manipulated, it should *lose* its taint.

To support both types of policies, we split the taint words in two parts: bits 0-15 which are *or*:ed together when combined (referred to as *or*-flags), and bits 16-31 which are *and*:ed together when combined (referred to as *and*-flags). Rather than changing the actual propagation code in TaintDroid however, we simply changed the default taint from 32 zeros to 16 zeros followed by 16 ones and inverted the interpretation of the *and*-flags.

7.2 Intercepting Calls using Monitor Inlining

Whenever a program is about to call a function the monitor needs to check if the call is allowed by the policy or not. If it is not allowed the call should be prevented (for instance by terminating the execution) and if it is allowed the label of the result of the function call should be set according to the policy. This task could be handled by the VM. However, our implementation delegates the task to the program itself by inlining the monitor code into the program. This approach is known as monitor inlining [11] and has the advantage of not requiring extra support from the execution environment.

Inlining a monitor into `P` involves the following steps:

1. Parse a policy $\mathcal{P}$ given in Definition 9 syntax.
2. Traverse `P`'s code and replace each call to a policy relevant function f_i with code that does the following
   (a) Copy the arguments from the operand stack to local variables (as they may be needed after the call when evaluating the label expression in step 2e)
   (b) Until a guard $guard_{ij}$ holds, evaluate the guards successively starting with $guard_{i1}$. If $guard_{ij}$ holds store j in a temporary variable x and go to 2d.
   (c) Terminate the execution due to policy violation.
   (d) Perform the original function call.
   (e) Evaluate $expr_{1x}$ and assign the resulting label to the value returned by the function.

Steps 2b and 2e rely on code for accessing the labels of certain values. Since the labels are not accessible directly through bytecode instructions this requires interaction with TaintDroid. As TaintDroid was not originally designed to interact with client programs, a few minor modifications to TaintDroid were required (exposing some internal methods).

The inlining step is fully automatic and can conveniently be added to the compile chain, as it has been done by editing the build settings in the Eclipse IDE for example.

7.2.1 Limitations due to Client-Side Inlining

The general client side inlining limitations has been explored previously [8, 7]. The main drawback in our setting is the lack of complete mediation [28] (function calls made internally by the runtime library are not observable).

The solution is to let the policy prevent internal computations from violating the policy, by restricting the calls performed by the client. For example, if `exec` is a policy relevant function, the policy must also restrict calls to functions that call `exec` internally, such as wrapper functions. This may require an over approximation of the intended policy.

Provided all internal policy violations are avoided by the above technique, as the resulting label would be overridden by the monitor when control returns to the client code regardless of the internal computations, the lack of complete mediation is irrelevant when calling functions which are explicitly mentioned in the policy. However, one can not expect that a policy has a clause for each function in the Java API. To relate the behavior of our implementation to the theory of the framework, the semantics of calling a method not mentioned in the policy is considered to be the same as if the body of that method was recursively unfolded into the client code. In other words, the labels of values returned by internal function calls are determined solely by the rules for arithmetic operations (as described in Section 7.1.1).

7.3 Handling Impure Functions

For simplicity, the theoretical presentation of the framework is restricted to pure functions. As shown in Theorem 1, the value of an argument in such setting is fully determined by its FAT. For this reason the observable actions do not entail information regarding actual values of arguments. However, a language like Java depends heavily on impure functions. The case studies highlight the importance of being able to reason about argument values.

The modifications needed for proper handling of impure functions are however straightforward and do not affect the theorems presented in the paper. FATs (and thus observable actions) need to take argument values into account which is done by the following T-AppFun reduction:

$$f^n\ \tau_1 : c_1\ \ \ldots\ \ \tau_n : c_n \xrightarrow{f(\tau_1:c_1,\ldots,\tau_n:c_n)} f(\tau_1 : c_1, \ldots, \tau_n : c_n) : [\![f^n(c_1, \ldots, c_n)]\!]$$

Similarly $\oplus$ needs to operate on labeled constants instead of just labels and the L-AppFun needs to be written as

$$\frac{\gamma = \mathsf{L}_f \oplus \alpha_1 : c_1 \oplus \cdots \oplus \alpha_n : c_n}{f^n\ \alpha_1 : c_1\ \ \ldots\ \ \alpha_n : c_n \to \gamma : [\![f^n(c_1, \ldots, c_n)]\!]}$$

These modifications allows the policy guards and expressions to refer to the argument values in addition to the argument labels. Modifications to the definitions of the derived policy automaton and labeling scheme are straightforward.

8. CASE STUDIES

This section evaluates the approach and the implementation in five case studies with varying characteristics. The webpage [24] contains full details including concrete policies.

8.1 Case Study 1: DroidLocator

Just as the popular application *Find My Phone* for iPhone, the *DroidLocator* application allows the user to locate a lost or stolen Android device through a web service. As opposed to Find My Phone and other similar services however, DroidLocator prevents server administrators and third parties from using the location data maliciously. It does so by *encrypting* the location data, based on a user-provided key, before uploading it to the server. When the user later retrieves the encrypted location data, he or she can decrypt it without revealing the location to anyone else.

Application DroidLocator is a small application written by one of the paper's authors. It retrieves the location data from the GPS hardware, uses the `javax.crypto` package to encrypt it with a key retrieved through `EditText.getText`, and submits it to the server using the standard socket API.

Policy The desired policy states that (A) the location may not be sent over the network unless it is encrypted and that (B) the encryption key needs to be provided by the user (retrieved through `EditText.getText` on an object with no prior calls to `EditText.setText`). Figure 5 shows the policy expressed in terms of the syntax in Definition 9. The formal semantics of this policy is provided by the STA, $\mathcal{A}_{DL}$, obtained from Figure 5 by Definition 10. Examples of FATs accepted and rejected by $\mathcal{A}_{DL}$ are found in Figure 6.

The labeling scheme used at runtime is obtained by following Definition 11 and adapting it to TreeDroid as described in Section 7. The resulting set of labels L is: $\{0^{16}1^{16}$ (L_0), $10^{15}1^{16}$ (sock), $010^{14}1^{16}$ (conf), $0010^{13}1^{16}$ (nonuser), $0^{17}1^{15}$ (userenc), $0^{16}101^{14}$ (userinp)$\}$, and the bitwise-or operator is used for $\oplus$.

$\{\mathsf{L}_0, \mathsf{sock}, \mathsf{conf}, \mathsf{userinp}, \mathsf{userenc}, \mathsf{nonuser}\}$,
$\{$`LocationManager.`

`getLastKnownLocation`(α):	$true \to \mathsf{conf}$
`Location.toString`(α_{loc}):	$true \to \alpha_{loc}$
`EditText.setText`$(\alpha_{et}, \alpha_{text})$:	$true \to \mathsf{nonuser}$
`EditText.getText`(α_{et}):	$\alpha_{et} \neq \mathsf{nonuser} \to \mathsf{userinp}$
	$true \to \mathsf{L}_0$
`Editable.toString`(α):	$true \to \alpha$
`SimpleCrypto.getRawKey`(α):	$true \to \alpha$
`SimpleCrypto.encrypt`(α_k, α_m):	$\alpha_k = \mathsf{user} \to \mathsf{userenc}$
`Socket.getOutputStream`(α):	$true \to \mathsf{sock}$
`OutputStream.write`$(\alpha_{out}, \alpha_{val})$:	$\alpha_{val} = \mathsf{L}_0 \to \mathsf{L}_0$
	$\alpha_{val} = \mathsf{userenc} \to \mathsf{L}_0$
	$\alpha_{out} \neq \mathsf{sock} \to \mathsf{L}_0\}$

Figure 5: Policy for DroidLocator case study.

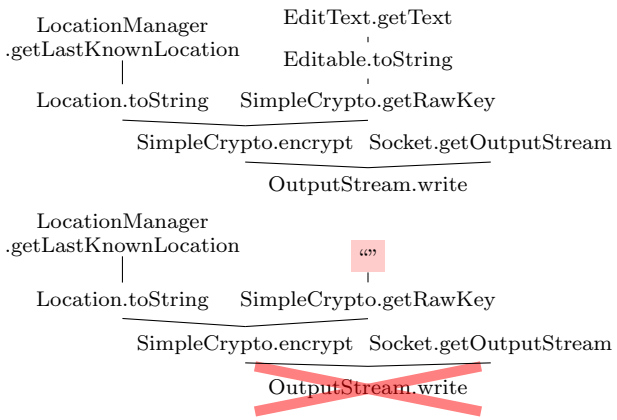

Figure 6: Accepted and rejected FATs

Results The behavior was unaffected by monitor inlining since the original application adhered to the policy. When the code was changed to use as encryption key a predefined string literal (such as, in Figure 6, the empty string), the execution was terminated before the location was uploaded.

8.2 Case Study 2: Sms2Group

An application, *Sms2Group*, requiring the SEND_SMS permission, allowing users to send SMS-messages to groups of contacts, is studied. The policy in this study restricts which numbers messages may be sent to.

Application Sms2Group has been developed for the purpose of this study. The application allows the user to automate the task of sending text messages to a group of contacts. It relies on the group attribute in the contact book, fetched using the content provider API and uses the ordinary `SmsManager.sendTextMessage` method to send SMSes.

Policy Messages are prevented from being sent to arbitrary numbers by ensuring that destination numbers (first argument of `sendTextMessage`) originate from the local address

Case study:	DroidLocator	Sms2Group	Bankdroid+HttpClient	Auto Birthday SMS	Lovetrap
Lines of Java source code:	330	240	101079	N/A	N/A
Size bytecode before inlining:	16.8 kB	17.0 kB	2.6 MB	193.9 kB	55.0 kB
Size increase due to inlining:	24.9 %	35.2 %	0.395 %	43.2 %	5.30 %
Inlining duration:	178 ms	190 ms	2740 ms	870 ms	210 ms
Policy relevant method calls:	11	15	213	359	1
Number of policy clauses:	9	4	14	5	4
Average total execution time:	142 ms	884 ms	9780 ms	N/A	N/A
Overhead due to TaintDroid:	38.9 %	53.3 %	28.0 %	N/A	N/A
Overhead due to inlined code:	45.4 %	16.9 %	19.5 %	N/A	N/A
Downloads on Google Play:	N/A	N/A	>100,000	>10,000	N/A

Figure 4: Statistics from the case studies.

book. The label specifying that a value is a valid destination number is encoded using an *and*-flag which prevents attackers from using a modified address book number. This general policy naturally separates legitimate executions from malicious ones. Using traditional inlining techniques this type of policy would be expressed using a guard that scans the address book and checks that the destination number is present. There are two conceptual differences between these approaches. As opposed to a policy that relies on scanning the address book, our policy expresses that an SMS may not be sent to numbers with arbitrary origin *even* if the number is present in the address book. In this sense our policy is stricter. Another difference is that a scan of the address book is typically a linear operation, whereas checking the taint of a value is a constant time operation.

Results The inlining did not affect the functionality of the original program as it adheres to the policy. When changing the code so that the program attempts to send an SMS to a hard-coded number or a number from the address book concatenated with an arbitrary string the policy is violated and the program is terminated as expected.

8.3 Case Study 3: Bankdroid

This case study examines an internet banking application, *Bankdroid*, which allows users to review account information from several different banks. The application has many security concerns as the information it handles (balances, recent transactions, etc) is usually considered confidential. The main objective of the case study is to demonstrate how standard security policies can be applied *transparently on real world honest applications*, while still blocking dishonest variants of the same applications.

Application Bankdroid (40k lines of code) is distributed through Google Play and is currently installed on 100.000+ devices [17]. It uses the Apache HttpClient library to communicate with the banks. To allow the policy to be expressed at the level of sockets (instead of at the level of the Apache HttpClient API), the library has been included in the client code base which adds another 60k lines of code.

Policy The policy is a Chinese-Wall policy which states that data received from host A may be sent back to host A but not to some other host B. As mentioned in the above paragraph, the policy is expressed at the level of sockets which makes it general and applicable to many other applications requiring Internet access.

Results The application was modified to leak the current balance of each bank account to a host controlled by a potential attacker. The policy was then inlined in the modified application. When the leak was about to take place, the inlined code successfully terminated the execution.

8.4 Case Study 4: Auto Birthday SMS

Auto Birthday SMS is a popular application distributed on Google Play which allows the user to automatically send SMS-messages to friends on their birthdays. It is free of charge but displays ads which are retrieved over the network. It requires the INTERNET and SEND_SMS permissions. Applications requiring this combination of permissions are interesting to study since trojans sending premium-rate SMS messages are relatively common [12] and could potentially transform the phone into an SMS spamming bot. As demonstrated in this case study, TreeDroid is useful even for honest coders in order to harden their applications by inlining generic security policies.

Application Application data, including numbers to send messages to, are stored in a SQLite database. The code turns out to be vulnerable to SQL-injection attacks which can be exploited by any application with permission to modify the address book data. The code calls `SQLiteDatabase.execSQL`, which updates the database, with an unsanitized query containing the name of a contact. The contact name should be sanitized by `DatabaseUtil.sqlEscapeString` before running the query.

Policy The policy applied is a general sanitize-before-query policy stating that a query passed to `execSQL` must be a string literal, a result of `sqlEscapeString` or a concatenation of such strings. The label used for sanitized values is encoded using an *and*-flag to ensure that the concatenation of sanitized and unsanitized strings are considered sanitized. An example of an accepted FAT is found in Figure 7. Omitting the call to `sqlEscapeString` would result in a tree which would be rejected by the policy.

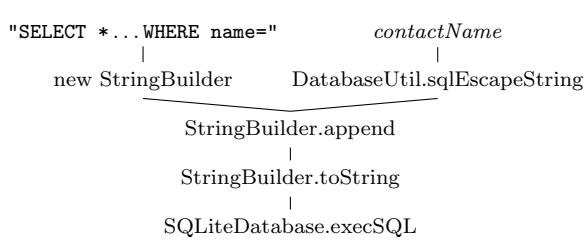

Figure 7: Accepted FAT for the SQL-policy.

Results The inlined code prevents the application from performing queries containing unsanitized arguments, such

as raw contact names, in the SQL statements. The original application violates the policy upon certain user actions, in such cases execution is successfully terminated by the monitor.

8.5 Case Study 5: Lovetrap

Lovetrap is a real world SMS-trojan detected by Symantec in July 2011 [32]. Among other bad behaviors, it sends premium rate SMS messages (which is the focus of this study). This case study demonstrates the efficiency of TreeDroid on real world attacks.

Application Lovetrap, which looks like a regular game, starts a service which downloads a list of numbers and messages which it repeatedly tries to send by SMS.

Policy The policy from case study 1 is reused without modification, which is an indication of the policy genericness.

Results By locally redirecting requests going to the host of the attacker to our own server, we managed to supervise the actions of the trojan. The monitor inlining at the bytecode level proceeds as expected without special tweaking. After inlining, the trojan's service is terminated immediately and therefore no longer able to send SMS messages as intended.

8.6 Statistics

Case studies statistics have been collected in Table 4. For applications where we have access to the source code, business logic execution time has been measured. In Bankdroid we measured the time it takes to update the accounts, for DroidLocator we measured the time it takes to encrypt and upload the location, and for Sms2Group we measured the time it takes to collect the group information and send the SMS. Taint tracking runtime overhead has been estimated by TaintDroid's authors to about 14 % on a Google Nexus One [10]. Our measurements (significantly higher, as expected since they are performed using Dalvik in debug mode on an Android emulator) are included for comparison with the runtime overhead due to the inlined code. The bytecode size overhead in the Auto Birthday SMS study is due to the fact that the relatively common operation of concatenating strings is considered policy relevant.

9. CONCLUSIONS AND FUTURE WORK

The paper presents a new monitoring technique using tree automata to track and enforce data processing constraints in a novel way. Many security properties, which were either difficult or impossible to express using existing techniques, can be treated. The approach is theoretically well-founded and practical as demonstrated by the various case studies.

Usability could be further increased by using techniques that give a formal semantics to textual policies [26, 33]. It would allow application authors to provide usage description of required permissions (which is a recommended good practice) that are both user-readable and from which enforceable formal policies could be extracted.

Focus on direct flows that can be tracked by taint analysis is not a fundamental limitation of the approach. A possible direction for future work would be to extend the program model, notion of observable actions and policy semantics to support indirect flows (decisions influencing data processing). It should be noted, however, that no practical approaches currently exists that can provide a comprehensive protection against covert flows anyway, and so it is far from clear that the added quality of protection offered by such an extension really motivates the additional complexity and runtime overhead.

Concurrency poses no problems if the order of policy relevant actions does not matter (as for local policies) since each thread can be monitored in isolation. For non-local policies, however, where the order of the actions does matter, the monitor has to synchronize the threads to exclude schedulings that yield illegal executions. This requires inlined code to be executed atomically together with policy relevant actions. This is problematic for a client-side inliner due to the fact that there is no way to acquire a lock before calling a method, and releasing it immediately when control has passed into the API method. The solution is either to release the lock after the policy relevant method has completed, i.e. use a *blocking inliner* [8] or restrict attention to so-called *race free* policies [6].

Leveraging tree automata theory allows for reuse of existing algorithms such as automata containment and minimization. Exploring these techniques further is left as future work.

Finally, our approach could very well be used in conjunction with existing control-flow bound techniques which would allow policies to express properties such as "*if the authenticate method returned true for credentials that has been provided by the user, queries do not have to be sanitized*". Some techniques for linear monitoring can also naturally be applied directly to tree based monitoring. Translating the idea of using edit automata instead of ordinary word automata into the context of tree automata would for instance allow us to express policies such as "*whenever an unsanitized query is about to be evaluated, sanitize it first*".

Acknowledgments We would like to express our appreciation to Tomas Andréasson for his work on the inliner tool and assistance in carrying out the case studies.

10. REFERENCES

[1] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *Proc. symp. Formal Methods*, FM'08, pages 262–277, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] K. Bierhoff and J. Aldrich. Plural: checking protocol compliance under aliasing. In *Companion of the intl. Conf. on Software engineering*, ICSE Companion '08, pages 971–972, New York, NY, USA, 2008. ACM.

[3] K. Bierhoff, N. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *LNCS*, pages 195–219. Springer Berlin / Heidelberg, 2009.

[4] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proc. work. Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[6] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining and certification for

multithreaded Java. To appear in Mathematical Structures in Computer Science.

[7] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded Java. In *European Conf. of Object-Oriented Computing*, pages 546–569. Springer-Verlag, July 2009.

[8] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably Correct Inline Monitoring for Multithreaded Java-like Programs. *Journal of Computer Security*, 18(1):37–59, 2010.

[9] R. Deline and M. Fahndrich. Typestates for objects. In *European Conf. of Object-Oriented Computing*, volume 3086 of *LNCS*, 2004.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. conf. Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[11] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, School of Computer Science, Reykjavík University, Ithaca, NY, USA, 2004. AAI3114521.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. work. Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[13] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. In *Transactions on Programming Languages and Systems*, pages 307–318. ACM Press, 2001.

[14] M. Gandhe, G. Venkatesh, and A. Sanyal. Labeled lambda-calculus and a generalized notion of strictness (an extended abstract). In *Proc., Concurrency and Knowledge*, ACSC '95, pages 103–110, London, UK, 1995. Springer-Verlag.

[15] I. Gartner. Gartner says sales of mobile devices in second quarter of 2011 grew 16.5 percent year-on-year; smartphone sales grew 74 percent. `http://www.gartner.com/it/page.jsp?id=1764714`. Accessed February 17, 2012.

[16] S. Ghoshal, S. Manimaran, G. Roşu, T. F. Şerbănuţă, and G. Ştefănescu. Monitoring IVHM systems using a monitor-oriented programming framework. In *The Sixth NASA Langley Formal Methods Workshop (LFM 2008)*, 2008.

[17] Google, Inc. Google play: Bankdroid. `https://play.google.com/store/apps/details?id=com.liato.bankdroid`. Accessed April 23, 2012.

[18] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proc. Annual Computer Security Applications Conf.*, pages 303–311, 2005.

[19] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Proc. work. Programming Languages and Analysis for Security*, pages 11–20, Tucson, Arizona, June 2008.

[20] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. work. Programming languages and analysis for security*, PLAS '06, pages 7–16, New York, NY, USA, 2006. ACM.

[21] J.-J. Lévy. *Réductions correctes et optimales dans le lambda calcul*. PhD thesis, Paris 7, 1978.

[22] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[23] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proc. of the European Symposium on Research in Computer Security*, Sept. 2010.

[24] A. Lundblad and T. Andréasson. TreeDroid: Tree Automaton Based Policy Inlining. `https://sites.google.com/site/treedroidcasestudies`. Accessed May 4, 2012.

[25] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.

[26] S. M. Montazeri, N. Roy, and G. Schneider. From Contracts in Structured English to CL Specifications. In *Proc. Work. Formal Languages and Analysis of Contract-Oriented Software*, volume 68 of *EPTCS*, pages 55–69, Málaga, Spain, Sept 2011.

[27] G. Roşu and S. Bensalem. Allen linear (interval) temporal logic –translation to LTL and monitor synthesis–. In *Proc. intl. conf. Computer Aided Verification*, volume 4144 of *LNCS*, pages 263–277. Springer, 2006.

[28] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278 – 1308, sept. 1975.

[29] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.

[30] K. Sen, G. Rosu, and G. Agha. Generating optimal linear temporal logic monitors by coinduction. In *Proc. Asian Computing Science Conference*, pages 260–275. Springer-Verlag, 2004.

[31] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.

[32] Symantec Corporation. Android.lovetrap. `http://www.symantec.com/security_response/writeup.jsp?docid=2011-072806-2905-99`. Accessed May 3, 2012.

[33] R. Thion and D. Le Métayer. FLAVOR: A formal language for a posteriori verification of legal rules. In *Proc. Symp. Policies for Distributed Systems and Networks*, pages 1–8. IEEE Computer Society, June 2011.

[34] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *Proc. Programming language design and implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[35] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *WOOT*, pages 81–90, 2011.

[36] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, Oct. 2000.