Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System

Hendra Gunadi[1] and Alwen Tiu[2]

[1] Research School of Computer Science, The Australian National University
[2] School of Computer Engineering, Nanyang Technological University

Abstract. We present a design and an implementation of a security policy specification language based on metric linear-time temporal logic (MTL). MTL features temporal operators that are indexed by time intervals, allowing one to specify timing-dependent security policies. The design of the language is driven by the problem of runtime monitoring of applications in mobile devices. A main case of the study is the privilege escalation attack in the Android operating system, where an app gains access to certain resource or functionalities that are not explicitly granted to it by the user, through indirect control flow. To capture these attacks, we extend MTL with recursive definitions, that are used to express call chains betwen apps. We then show how the metric operators of MTL, in combination with recursive definitions, can be used to specify policies to detect privilege escalation, under various fine grained constraints. We present a new algorithm, extending that of linear time temporal logic, for monitoring safety policies written in our specification language. The monitor does not need to store the entire history of events generated by the apps, something that is crucial for practical implementations. We modified the Android OS kernel to allow us to insert our generated monitors modularly. We have tested the modified OS on an actual device, and show that it is effective in detecting policy violations.

1 Introduction

Android is a popular mobile operating system (OS) that has been used in a range of mobile devices such as smartphones and tablet computers. It uses Linux as the kernel, which is extended with an application framework (middleware). Most applications of Android are written to run on top of this middleware, and most of Android-specific security mechanisms are enforced at this level.

Android treats each application as a distinct user with a unique user ID. At the kernel level, access control is enforced via the standard Unix permission mechanism based on the user id (and group id) of the app. At the middleware level, each application is sandboxed, i.e., it is running in its own instance of Dalvik virtual machine, and communication and sharing between apps are allowed only through an inter-process communication (IPC) mechanism. Android middleware provides a list of resources and services such as sending SMS, access

to contacts, or internet access. Android enforces access control to these services via its permission mechanism: each service/resource is associated with a certain unique permission tag, and each app must request permissions to the services it needs at installation time. Everytime an app requests access to a specific service/resource, Android runtime security monitor checks whether the app has the required permission tags for that particular service/resource. A more detailed discussion of Android security architecture can be found in [14].

One problem with Android security mechanism is the problem of *privilege escalation*, that is, the possibility of an app to gain access to services or resources that it does not have permissions to access. Obviously privilege escalation is a common problem of every OS, e.g., when a kernel bug is exploited to gain root access. However, in Android, privilege escalation is possible even when apps are running in the confine of Android sandboxes [21, 10, 8]. There are two types of attacks that can lead to privilege escalation [8]: the *confused deputy attack* and the *collusion attack*. In the confused deputy attack, a legitimate app (the deputy) has permissions to certain services, e.g., sending SMS, and exposes an interface to this functionality without any guards. This interface can then be exploited by a malicious app to send SMS, even though the malicious app does not have the permission. Recent studies [21, 17, 9] show some system and consumer apps expose critical functionalities that can be exploited to launch confused deputy attacks. The collusion attack requires two or more malicious apps to collaborate. We have yet to encounter such a malware, either in the Google Play market or in the third party markets, although a proof-of-concept malware with such properties, called SoundComber [24], has been constructed.

Several security extensions to Android have been proposed to deal with privilege escalation attacks [12, 15, 8]. Unlike these works, we aim at designing a high-level policy language that is expressive enough to capture privilege escalation attacks, but is also able to express more refined policies (see Section 4). Moreover, we aim at designing a lightweight monitoring framework, where policy specifications can be modified easily and enforced efficiently. Thus we aim at an automated generation of security monitors that can efficiently enforce policies written in our specification language.

On the specific problem of detecting privilege escalation, it is essentially a problem of tracking (runtime) control flow, which is in general a difficult problem and would require a certain amount of static analysis [11, 13]. So we adopt a 'lightweight' heuristic to ascertain causal dependency between IPC calls: we consider two successive calls, say from A to B, followed by a call from B to C, as causally dependent if they happen within a certain reasonably short time frame. This heuristic can be easily circumvented if B is a colluding app. So the assumption that we make here is that B is honest, i.e., the confused deputy. For example, a privilege escalation attack mentioned in [21] involves a malicious app, with no permission to access internet, using the built-in browser (the deputy) to communicate with a server. In our model, the actual connection (i.e., the network socket) is treated as virtual app, so the browser here acts as a deputy that calls (opens) the network socket on behalf of the malicious app. In such a scenario, it

is reasonable to expect that the honest deputy would not intentionally delay the opening of sockets. So our heuristic seems sensible in the presence of confused deputy attacks, but can be of course circumvented by colluding apps (collusion attacks). There is probably no general solution to detect collusion attacks that can be effective in all cases, e.g., when covert channels are involved [24], so we shall restrict to addressing the confused deputy attacks.

The core of our policy language, called RMTL, is essentially a past-fragment of metric linear temporal logic (MTL) [1, 26, 2]. We consider only the fragment of MTL with past-time operators, as this is sufficient for our purpose to enforce history-sensitive access control. This also means that we can only enforce some, but not all, safety properties [20], e.g., policies capturing obligations as in, e.g., [2], cannot be enforced in our framework. Temporal operators are useful in this setting to enforce access control on apps based on histories of their executions; see Section 4. Such a history-dependent policy cannot be expressed in the policy languages used in [12, 15, 8].

MTL by itself is, however, insufficient to express transitive closures of relations, which is needed to specify IPC call chains between apps, among others. To deal with this, we extend MTL with recursive definitions, e.g., one would be able to write a definition such as:

$$trans(x, y) \coloneqq call(x, y) \vee \exists z.\, \diamondsuit_n\, trans(x, z) \wedge call(z, y), \tag{1}$$

where *call* denotes the IPC event, and x, y, z denote the apps. This equation defines *trans* as the transitive closure of *call*. The metric operator $\diamondsuit_n \phi$ means intuitively ϕ holds within n time units in the past; we shall see a more precise definition of the operators in Section 2. Readers familiar with modal μ-calculus [6] will note that this is but a syntactic sugar for μ-expressions for (least) fixed points.

To be practically enforceable in Android, RMTL monitoring algorithm must satisfy an important constraint, i.e., the algorithm must be *trace-length independent* [5]. This is because the number of events generated by Android can range in the thousands per hour, so if the monitor must keep all the events generated by Android, its performance will degrade significantly over time. Another practical consideration also motivates a restriction to metric operators that we adopt in RMTL. More specifically, MTL allows a metric version of the 'since' operator of the form $\phi_1\ \mathbb{S}_{[m,n)}\ \phi_2$, where $[m, n)$ specifies a half-closed (discrete) time interval from m to n. The monitoring algorithm for MTL in [26] works by first expanding this formula into formulas of the form $\phi_1\ \mathbb{S}_{[m',n')}\ \phi_2$ where $[m', n')$ is a moving window of the interval (with the minimum value of 0). A similar expansion is also used implicitly in monitoring for first-order MTL in [2], i.e., in their incremental automatic structure extension in their first-order logic encoding for the 'since' and 'until' operators. In general, if we have k nested occurrences of metric operators, each with interval $[m, n)$, the number of formulas produced by this expansion is bounded by $O(n^k)$ In Android, event timestamps are in milliseconds, so this kind of expansion is not practically feasible. For example, suppose we have a policy that monitors three successive IPC calls that happen within 10

seconds between successive calls. This requires two nested metric operators with intervals $[0, 10^4)$ to specify. The above naive expansion would produce around 10^8 formulas, and assuming the truth value of each formula is represented with 1 bit, this would require around 100 MB of storage to store all their truth values, something which is not preferable in the setting of smartphones where storage is limited.

An improvement to the expansion mentioned above is proposed in [3, 23], where one keeps a sequence of timestamps for each metric temporal operator occuring in the policy. This solution, although avoids the exponential expansion, is strictly speaking not trace-length independent. This solution seems optimal so it is hard to improve it without further restriction to the policy language. We show that, if one restricts the intervals of metric operators to the form $[0, n)$, one only needs to keep one timestamp for each metric operator in monitoring; see Section 3.

To summarise, our contributions are as follows:

1. In terms of results in runtime verification, our contribution is in the design of a new logic-based policy language that extends MTL with recursive definitions, that avoids exponential expansion of metric operators, and for which the policy enforcement is trace-length independent. In [5], a policy language based on first-order LTL and a general monitoring algorithm are given, but they do not allow recursive definitions nor metric operators. Such definitions and operators could perhaps be encoded using first-order constructs (e.g., encoding recursion via Horn clauses, and define timestamps explicitly as a predicate), but the resulting monitoring procedure is not guaranteed to be trace-length independent.
2. In terms of the application domain, ours is the first implementation of a logic-based runtime security monitor for Android that can enforce history-based access control policies, including those that concern privilege escalations. Our monitoring framework can express temporal and metric-based policies not possible in existing works [12, 15, 8].

The rest of the paper is organized as follows. Section 2 introduces our policy language RMTL. In Section 3, we present the monitoring algorithm for RMTL and state its correctness. Some example policies are described in 4. Section 5 discusses our implementation of the monitors for RMTL, and the required modification of Android OS kernel to integrate our monitor into the OS. In Section 6 we conclude and discuss related and future works. Detailed proofs of the lemmas and theorems are omitted here but can be found in [18]. Details of the implementation of the monitor generator and the binaries of the modified Android OS are available online.[3]

[3] http://users.cecs.anu.edu.au/~hengunadi/LogicDroid.html.

2 The policy specification language RMTL

Our policy specification language, which we call RMTL, is based on an extension of metric linear-time temporal logic (MTL) [25]. The semantics of LTL [22] is defined in terms of models which are sequences of states (or worlds). In our case, we restrict to finite sequences of states. MTL extends LTL models by adding timestamps to each state, and adding temporal operators that incorporate timing constraints, e.g., MTL features temporal operators such as $\Diamond_{[0,3)}\phi$ which expresses that ϕ holds in some state in the future, and the timestamp of that world is within 0 to 3 time units from the current timestamp. We restrict to a model of MTL that uses discrete time, i.e., timestamps in this case are non-negative integers. We shall also restrict to the past-time fragment of MTL.

We extend MTL with two additional features: first-order quantifiers and recursive definitions. Our first-order language is a multi-sorted one. We consider only two sorts, which we call *prop* (for 'properties') and *app* (for denoting applications). Sorts are ranged over by α. We first fix a *signature* Σ for our first-order language, which is used to express terms and predicates of the language. We consider only constant symbols and predicate symbols, but no function symbols. We distinguish two types of predicate symbols: *defined* predicates and *undefined* ones. The defined predicate symbols are used to write recursive definitions and to each of such symbols we associate a formula as its definition.

Constant symbols are ranged over by a, b and c, undefined predicate symbols are ranged over by p, q and r, and defined predicate symbols are ranged over by P, Q and R. We assume an infinite set of sorted variables $\mathcal{V}$, whose elements are ranged over by x, y and z. We sometimes write x_α to say that α is the sort of variable x. A *Σ-term* is either a constant symbol $c \in \Sigma$ or a variable $x \in \mathcal{V}$. We use s, t and u to range over terms. To each symbol in Σ we associate a sort information. We shall write $c : \alpha$ when c is a constant symbol of sort α. A predicate symbol of arity n has sort of the form $\alpha_1 \times \cdots \times \alpha_n$, and such a predicate can only be applied to terms of sorts $\alpha_1, \ldots, \alpha_n$.

Constant symbols are used to express permissions in the Android OS, e.g., reading contacts, sending SMS, etc., and user ids of apps. Predicate symbols are used to express events such as IPC calls between apps, and properties of an app, such as whether it is a system app, a trusted app (as determined by the user). As standard in first-order logic (see e.g. [16]), the semantics of terms and predicates are given in terms of a first-order structure, i.e., a set $\mathcal{D}_\alpha$, called a *domain*, for each sort α, and an interpretation function I assigning each constant symbol $c : \alpha \in \Sigma$ an element of $c^I \in \mathcal{D}_\alpha$ and each predicate symbol $p : \alpha_1 \times \cdots \times \alpha_n \in \Sigma$ an n-ary relation $p^I \subseteq \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n}$. We shall assume constant domains in our model, i.e., every world has the same domain.

The formulas of RMTL is defined via the following grammar:

$$F \coloneqq \bot \mid p(t_1, \ldots, t_m) \mid P(t_1, \ldots, t_n) \mid F \vee F \mid \neg F \mid \bullet F \mid F \mathbin{\mathbb{S}} F \mid \blacklozenge F \mid \circledcirc F \mid \bullet_n F \mid F \mathbin{\mathbb{S}_n} F \mid \blacklozenge_n F \mid \circledcirc_n F \mid \exists_\alpha x.F$$

where m and n are natural numbers. The existential quantifier is annotated with a sort information α. For most of our examples and applications, we only quan-

tify over variables of sort *app*. The operators indexed by n are *metric temporal operators*. The $n \geq 1$ here denotes the interval $[0, n)$, so these are special cases of the more general MTL operators in [25], where intervals can take the form $[m, n)$, for $n \geq m \geq 0$. We use ϕ, φ and ψ to range over formulas. We assume that unary operators bind stronger than the binary operators, so $\bullet\phi \vee \psi$ means $(\bullet\phi) \vee \psi$. We write $\phi(x_1, \ldots, x_n)$ to denote a formula whose free variables are among $x_1, \ldots, x_n$. Given such a formula, we write $\phi(t_1, \ldots, t_n)$ to denote the formula obtained by replacing x_i with t_i for every $i \in \{1, \ldots, n\}$.

To each defined predicate symbol $P : \alpha_1 \times \cdots \times \alpha_n$, we associate a formula ϕ_P, which we call the *definition* of P. Notationally, we write $P(x_1, \ldots, x_n) \coloneqq \phi_p(x_1, \ldots, x_n)$. We require that ϕ_P is *guarded*, i.e., every occurrence of any recursive predicate Q in ϕ_P is prefixed by either $\bullet$, $\bullet_m$, $\diamondsuit$ or $\diamondsuit_n$. This guardedness condition is important to guarantee termination of recursion in model checking.

Given the above logical operators, we can define additional operators via their negation, e.g., $\top$ is defined as $\neg\bot$, $\phi \wedge \psi$ is defined as $\neg(\neg\phi \vee \neg\psi)$, $\phi \to \psi$ is defined as $\neg\phi \vee \psi$, and $\forall_\alpha x.\phi$ is defined as $\neg(\exists_\alpha x.\neg\phi)$, etc.

Before proceeding to the semantics of RMTL, we first define a well-founded ordering on formulae of RMTL, which will be used later.

Definition 1. *We define a relation $\prec_S$ on the set RMTL formulae as the smallest relation satisfying the following conditions:*

1. *For any formula ϕ of the form $p(\vec{t})$, $\bot$, $\bullet\psi$, $\bullet_n\psi$, $\diamondsuit\psi$ and $\diamondsuit_n\psi$, there is no ϕ' such that $\phi' \prec_S \phi$.*
2. *For every recursive definition $P(\vec{x}) \coloneqq \phi_P(\vec{x})$, we have $\phi_P(\vec{t}) \prec_S P(\vec{t})$ for every terms $\vec{t}$.*
3. *$\psi \prec_S \psi \vee \psi'$, $\psi \prec_S \psi' \vee \psi$, $\psi \prec_S \neg\psi$, and $\psi \prec_S \exists x.\psi$.*
4. *$\psi_i \prec_S \psi_1 \mathbb{S} \psi_2$, and $\psi_i \prec_S \psi_1 \mathbb{S}_n \psi_2$, for $i \in \{1, 2\}$*

We denote with $\prec$ the reflexive and transitive closure of $\prec_S$.

Lemma 1. *The relation $\prec$ on RMTL formulas is a well-founded partial order.*

For our application, we shall restrict to finite domains. Moreover, we shall restrict to an interpretation I which is injective, i.e., mapping every constant c to a unique element of $\mathcal{D}_\alpha$. In effect we shall be working in the term model, so elements of $\mathcal{D}_\alpha$ are just constant symbols from Σ. So we shall use a constant symbol, say $c : \alpha$, to mean both $c \in \Sigma$ and $c^I \in \mathcal{D}_\alpha$. With this fix interpretation, the definition of the semantics (i.e., the satisfiability relation) can be much simplified, e.g., we do not need to consider valuations of variables. A *state* is a set of undefined atomic formulas of the form $p(c_1, \ldots, c_n)$. Given a sequence σ, we write $|\sigma|$ to denote its length, and we write σ_i to denote the i-th element of σ when it is defined, i.e., when $1 \leq i \leq |\sigma|$. A *model* is a pair (π, τ) of a sequence of *states* π and a sequence of *timestamps*, which are natural numbers, such that $|\pi| = |\tau|$ and $\tau_i \leq \tau_j$ whenever $i \leq j$.

Let $<$ denote the total order on natural numbers. Then we can define a well-order on pairs (i, ϕ) of natural numbers and formulas by taking the lexicographical ordering $(<, \prec)$. The satisfiability relation between a model $\rho = (\pi, \tau)$,

a *world* $i \geq 1$ (which is a natural number) and a *closed* formula ϕ (i.e., ϕ contains no free variables), written $(\rho, i) \vDash \phi$, is defined by induction on the pair (i, ϕ) as follows, where we write $(\rho, i) \nvDash \phi$ when $(\rho, i) \vDash \phi$ is false.

- $(\rho, i) \nvDash \bot$
- $(\rho, i) \vDash \neg\phi$ iff $(\rho, i) \nvDash \phi$.
- $(\rho, i) \vDash p(c_1, \ldots, c_n)$ iff $p(c_1, \ldots, c_n) \in \pi_i$.
- $(\rho, i) \vDash P(c_1, \ldots, c_n)$ iff $(\rho, i) \vDash \phi(c_1, \ldots, c_n)$ where $P(\vec{x}) \coloneqq \phi(\vec{x})$.
- $(\rho, i) \vDash \phi \vee \psi$ iff $(\rho, i) \vDash \phi$ or $(\rho, i) \vDash \psi$.
- $(\rho, i) \vDash \bullet\phi$ iff $i > 1$ and $(\rho, i-1) \vDash \phi$.
- $(\rho, i) \vDash \blacklozenge\phi$ iff there exists $j \leq i$ s.t. $(\rho, j) \vDash \phi$.
- $(\rho, i) \vDash \diamondsuit\phi$ iff $i > 1$ and there exists $j < i$ s.t. $(\rho, j) \vDash \phi$.
- $(\rho, i) \vDash \phi_1 \mathbb{S} \phi_2$ iff there exists $j \leq i$ such that $(\rho, j) \vDash \phi_2$ and $(\rho, k) \vDash \phi_1$ for every k s.t. $j < k \leq i$.
- $(\rho, i) \vDash \bullet_n\phi$ iff $i > 1$, $(\rho, i-1) \vDash \phi$ and $\tau_i - \tau_{i-1} < n$.
- $(\rho, i) \vDash \blacklozenge_n\phi$ iff there exists $j \leq i$ s.t. $(\rho, j) \vDash \phi$ and $\tau_i - \tau_j < n$.
- $(\rho, i) \vDash \diamondsuit_n\phi$ iff $i > 1$ and there exists $j < i$ s.t. $(\rho, j) \vDash \phi$ and $\tau_i - \tau_j < n$.
- $(\rho, i) \vDash \phi_1 \mathbb{S}_n \phi_2$ iff there exists $j \leq i$ such that $(\rho, j) \vDash \phi_2$, $(\rho, k) \vDash \phi_1$ for every k s.t. $j < k \leq i$, and $\tau_i - \tau_j < n$.
- $(\rho, i) \vDash \exists_\alpha x.\phi(x)$ iff there exists $c \in \mathcal{D}_\alpha$ s.t. $(\rho, i) \vDash \phi(c)$.

Note that due to the guardedness condition in recursive definitions, our semantics for recursive predicates is much simpler than the usual definition as in μ-calculus, which typically involves the construction of a (semantic) fixed point operator. Note also that some operators are redundant, e.g., $\blacklozenge\phi$ can be defined as $\top \mathbb{S} \phi$, and $\diamondsuit\phi$ can be defined as $\bullet\blacklozenge\phi$. This holds for some metric operators, e.g., $\blacklozenge_n\phi$ and $\diamondsuit_n\phi$ can be defined as, respectively, $\top \mathbb{S}_n \phi$ and

$$\diamondsuit_n\phi = \bigvee_{i+j=n} \bullet_i \blacklozenge_j \phi \tag{2}$$

This operator will be used to specify an active call chain, as we shall see later, so it is convenient to include it in our policy language.

In the next section, we shall assume that $\blacklozenge$, $\diamondsuit$, $\blacklozenge_n$ as derived connectives. Since we consider only finite domains, $\exists_\alpha x.\phi(x)$ can be reduced to a big disjunction $\bigvee_{c \in \mathcal{D}_\alpha} \phi(c)$, so we shall not treat $\exists$-quantifier explicitly. This can be problematic if the domain of quantification is big, as it suffers the same kind of exponential explosion as with the expansion of metric operators in MTL [26]. We shall defer the explicit treatment of quantifiers to future work.

3 Trace-length independent monitoring

The problem of monitoring is essentially a problem of model checking, i.e., to decide whether $(\rho, i) \vDash \phi$, for any given $\rho = (\pi, \tau)$, i and ϕ. In the context of Android runtime monitoring, a state in π can be any events of interest that one would like to capture, e.g., the IPC call events, queries related to location information or contacts, etc. To simplify discussions, and because our main interest

is in privilege escalation through IPC, the only type of event we consider in π is the IPC event, which we model with the predicate $call : app \times app$.

Given a policy specification ϕ, a naive monitoring algorithm that enforces this policy would store the entire event history π and every time a new event arrives at time t, it would check $(([\pi; e], [\tau; t]), |\rho| + 1) \vDash \phi$. This is easily shown decidable, but is of course rather inefficient. In general, the model checking problem for RMTL (with finite domains) can be shown PSPACE hard following the same argument as in [4]. A design criteria of RMTL is that enforcement of policies does not depend on the length of history of events, i.e., at any time the monitor only needs to keep track of a fixed number of states. Following [5], we call a monitoring algorithm that satisfies this property *trace-length independent.*

For PTLTL, trace-length independent monitoring algorithm exists, e.g., the algorithm by Havelund and Rosu [19], which depends only on two states in a history. That is, satisfiability of $(\rho, i+1) \vDash \phi$ is a boolean function of satisfiability of $(\rho, i+1) \vDash \psi$, for every strict subformula ψ of ϕ, and satisfiability of $(\rho, i) \vDash \psi'$, for every subformula ψ' of ϕ. This works for PTLTL because the semantics of temporal operators in PTLTL can be expressed in a recursive form, e.g., the semantics of $\mathbb{S}$ can be equally expressed as [19]: $(\rho, i+1) \vDash \phi_1 \ \mathbb{S} \ \phi_2$ iff $(\rho, i+1) \vDash \phi_2$, or $(\rho, i+1) \vDash \phi_1$ and $(\rho, i) \vDash \phi_1 \ \mathbb{S} \ \phi_2$. This is not the case for MTL. For example, satisfiability of the unrestricted 'since' operator $\mathbb{S}_{[m,n)}$ can be equivalently expressed as:

$$(\rho, i+1) \vDash \phi_1 \ \mathbb{S}_{[m,n)} \ \phi_2 \text{ iff } \begin{array}{l} m = 0, n > 1, \text{ and } (\rho, i+1) \vDash \phi_2, \text{ or} \\ (\rho, i+1) \vDash \phi_1 \text{ and } (\rho, i) \vDash \phi_1 \ \mathbb{S}_{[m',n')} \ \phi_2 \end{array} \tag{3}$$

where $m' = min(0, m - \tau_{i+1} + \tau_i)$ and $n' = min(0, n - \tau_{i+1} + \tau_i)$. Since τ_{i+1} can vary, the value of m' and n' can vary, depending on the history ρ. We avoid the expansion of metric operators in monitoring by restricting the intervals in the metric operators to the form $[0, n)$. We show that clause (3) can be brought back to a purely recursive form. The key to this is the following lemma:

Lemma 2 (Minimality). *If $(\rho, i) \vDash \phi_1 \ \mathbb{S}_n \ \phi_2$ ($(\rho, i) \vDash \diamondsuit_n \phi$) then there exists an $m \leq n$ such that $(\rho, i) \vDash \phi_1 \ \mathbb{S}_m \ \phi_2$ (resp. $(\rho, i) \vDash \diamondsuit_m \phi$) and such that for every k such that $0 < k < m$, we have $(\rho, i) \nvDash \phi_1 \ \mathbb{S}_k \ \phi_2$ (resp., $(\rho, i) \nvDash \diamondsuit_k \phi$).*

Given ρ, i and ϕ, we define a function $\mathfrak{m}$ as follows:

$$\mathfrak{m}(\rho, i, \phi) = \begin{cases} m, \text{ if } \phi \text{ is either } \phi_1 \ \mathbb{S}_n \ \phi_2 \text{ or } \diamondsuit_n \phi' \text{ and } (\rho, i) \vDash \phi, \\ 0, \text{ otherwise.} \end{cases}$$

where m is as given in Lemma 2; we shall see how its value is calculated in Algorithm 3. The following theorem follows from Lemma 2.

Theorem 1 (Recursive forms). *For every model ρ, every $n \geq 1$, ϕ, ϕ_1 and ϕ_2, and every $1 < i \leq |\rho|$, the following hold:*

1. *$(\rho, i) \vDash \phi_1 \ \mathbb{S}_n \ \phi_2$ iff $(\rho, i) \vDash \phi_2$, or $(\rho, i) \vDash \phi_1$ and $(\rho, i-1) \vDash \phi_1 \ \mathbb{S}_n \ \phi_2$ and $n - (\tau_i - \tau_{i-1}) \geq \mathfrak{m}(\rho, i-1, \phi_1 \ \mathbb{S}_n \ \phi_2)$.*

Algorithm 1 $Monitor(\rho, i, \phi)$

$Init(\rho, \phi, prev, cur, mprev, mcur)$
for $j = 1$ to i **do**
  $Iter(\rho, j, \phi, prev, cur, mprev, mcur)$;
end for
return $cur[idx(\phi)]$;

Algorithm 2 $Init(\rho, \phi, prev, cur, mprev, mcur)$

for $k = 1, \ldots, m$ **do**
  $prev[k] := false$, $mprev[k] := 0$ and $mcur[k] := 0$;
end for
for $k = 1, \ldots, m$ **do**
  switch $(\phi_k = \bot)$
  case $(\bot)$**:** $cur[k] := false$;
  case $(p(\vec{c}))$**:** $cur[k] := p(\vec{c}) \in \pi_1$;
  case $(P(\vec{c}))$**:** $cur[k] := cur[idx(\phi_P(\vec{c}))]$; {Suppose $P(\vec{x}) := \phi_P(\vec{x})$.}
  case $(\neg\psi)$**:** $cur[k] := \neg cur[idx(\psi)]$;
  case $(\psi_1 \vee \psi_2)$**:** $cur[k] := cur[idx(\psi_1)] \vee cur[idx(\psi_2)]$;
  case $(\bullet\psi)$**:** $cur[k] := false$;
  case $(\diamondsuit\psi)$**:** $cur[k] := false$;
  case $(\psi_1 \mathbb{S} \psi_2)$**:** $cur[k] := cur[idx(\psi_2)]$;
  case $(\bullet_n\psi)$**:** $cur[k] := false$;
  case $(\diamondsuit_n\psi)$**:** $cur[k] := false; mcur[k] := 0$;
  case $(\phi_k = \psi_1 \mathbb{S}_n \psi_2)$**:**
    $cur[k] := cur[idx(\psi_2)]$;
    if $cur[k] = true$ **then** $mcur[k] := 1$;
    else $mcur[k] := 0$;
    end if
  end switch
end for
return $cur[idx(\phi)]$;

2. $(\rho, i) \vDash \diamondsuit_n\phi$ *iff* $(\rho, i-1) \vDash \phi$ *and* $\tau_i - \tau_{i-1} < n$, *or* $(\rho, i-1) \vDash \diamondsuit_n\phi$ *and* $n - (\tau_i - \tau_{i-1}) \geq \mathfrak{m}(\rho, i-1, \diamondsuit_n\phi)$.

Given Theorem 1, the monitoring algorithm for PTLTL in [19] can be adapted, but with an added data structure to keep track of the function $\mathfrak{m}$. In the following, given a formula ϕ, we assume that $\exists$, $\blacklozenge$ and $\diamondsuit$ have been replaced with its equivalent form as mentioned in Section 2.

Given a formula ϕ, let $Sub(\phi)$ be the set of subformulas of ϕ. We define a closure set $S^*(\phi)$ of ϕ as follows: Let $Sub^0(\phi) = Sub(\phi)$, and let

$$Sub^{n+1}(\phi) = Sub_n(\phi) \cup \{Sub(\phi_P(\vec{c})) \mid P(\vec{c}) \in Sub_n(\phi), \text{ and } P(\vec{x}) := \phi_P(\vec{x})\}$$

and define $Sub^*(\phi) = \bigcup_{n \geq 0} Sub^n(\phi)$. Since $\mathcal{D}_\alpha$ is finite, $Sub^*(\phi)$ is finite, although its size is exponential in the arities of recursive predicates. For our specific applications, the predicates used in our sample policies have at most arity of two

Algorithm 3 $Iter(\rho, i, \phi, prev, cur, mprev, mcur)$

Require: $i > 1$.
  $prev := cur$; $mprev := mcur$;
  for $k = 1$ to m **do** $mcur[k] := 0$; **end for**
  for $k = 1$ to m **do**
    switch (ϕ_k)
    case $(\bot)$**:** $cur[k] := false$;
    case $(p(\vec{c}))$**:** $cur[k] := p(\vec{c}) \in \pi_i$;
    case $(\neg\psi)$**:** $cur[k] := \neg cur[idx(\psi)]$;
    case $(P(\vec{c}))$**:** $cur[k] := cur[idx(\phi_P(\vec{c}))]$; {Suppose $P(\vec{x}) := \phi_P(\vec{x})$.}
    case $(\psi_1 \vee \psi_2)$**:** $cur[k] := cur[idx(\psi_1)] \vee cur[idx(\psi_2)]$;
    case $(\bullet\psi)$**:** $cur[k] := prev[idx(\psi)]$;
    case $(\diamondsuit\psi)$**:** $curr[k] := prev[idx(\psi)] \vee prev[\diamondsuit\psi]$;
    case $(\psi_1 \mathbb{S} \psi_2)$**:** $cur[k] := cur[idx(\psi_2)] \vee (cur[idx(\psi_1)] \wedge prev[idx(\psi_2)])$;
    case $(\bullet_n\psi)$**:** $cur[k] := prev[\psi] \wedge (\tau_i - \tau_{i-1} < n)$;
    case $(\diamondsuit_n\psi)$**:**
      $l := prev[idx(\psi)] \wedge (\tau_i - \tau_{i-1} < n)$;
      $r := prev[idx(\diamondsuit_n\psi)] \wedge (n - (\tau_i - \tau_{i-1}) \geq mprev[k]))$;
      $cur[k] := l \vee r$;
      if l **then** $mcur[k] := \tau_i - \tau_{i-1} + 1$;
      else if r **then** $mcur[k] := mprev[k] + \tau_i - \tau_{i-1}$;
      else $mcur[k] := 0$;
      end if
    case $(\psi_1 \mathbb{S}_n \psi_2)$**:**
      $l := cur[idx(\psi_2)]$;
      $r := cur[idx(\psi_1)] \wedge prev[k] \wedge (n - (\tau_i - \tau_{i-1}) \geq mprev[k])$;
      $cur[k] := l \vee r$;
      if l **then** $mcur[k] := 1$;
      else if r **then** $mcur[k] := mprev[k] + \tau_i - \tau_{i-1}$;
      else $mcur[k] := 0$;
      end if
    end switch
  end for
  return $cur[idx(\phi)]$;

(for tracking transitive calls), so this is still tractable. In future work, we plan to investigate ways of avoiding this explicit expansion of recursive predicates.

We now describe how monitoring can be done for ϕ, given ρ and $1 \leq i \leq |\rho|$. We assume implicitly a preprocessing step where we compute $Sub^*(\phi)$; we do not describe this step here but it is quite straightforward. Let $\phi_1, \phi_2, \ldots, \phi_m$ be an enumeration of $Sub^*(\phi)$ respecting the partial order $\prec$, i.e., if $\phi_i \prec \phi_j$ then $i \leq j$. Then we can assign to each $\psi \in Sub^*(\phi)$ an index i, s.t., $\psi = \phi_i$ in this enumeration. We refer to this index as $idx(\psi)$. We maintain two boolean arrays $prev[1, \ldots, m]$ and $cur[1, \ldots, m]$. The intention is that given ρ and $i > 1$, the value of $prev[k]$ corresponds to the truth value of the judgment $(\rho, i-1) \vDash \phi_k$ and the truth value of $cur[k]$ corresponds to the truth value of the judgment $(\rho, i) \vDash \phi_k$. We also maintain two integer arrays $mprev[1, \ldots, m]$ and

$mcur[1,\ldots,m]$ to store the value of the function $\mathfrak{m}$. The value of $mprev[k]$ corresponds to $\mathfrak{m}(\rho, i-1, \phi_k)$, and $mcur[k]$ corresponds to $\mathfrak{m}(\rho, i, \phi_k)$. Note that this preprocessing step only needs to be done once, i.e., when generating the monitor codes for a particular policy, which is done offline, prior to inserting the monitor into the operating system kernel.

The main monitoring algorithm is divided into two subprocedures: the initialisation procedure (Algorithm 2) and the iterative procedure (Algorithm 3). The monitoring procedure (Algorithm 1) is then a simple combination of these two. We overload some logical symbols to denote operators on boolean values. In the actual implementation, we do not actually implement the loop in Algorithm 1; rather it is implemented as an event-triggered procedure, to process each event as they arrive using $Iter$.

Theorem 2. $(\rho, i) \vDash \phi$ *iff* $Monitor(\rho, i, \phi)$ *returns true.*

The $Iter$ function only depends on two worlds: ρ_i and ρ_{i-1}, so the algorithm is trace-length independent. In principle there is no upperbound to its space complexity, as the timestamp τ_i can grow arbitrarily large, as is the case in [3]. Practically, however, the timestamps in Android are stored in a fixed size data structure, so in such a case, when the policy is fixed, the space complexity is constant (i.e., independent of the length of history ρ).

4 Examples

We provide some basic policies as an example of how we can use this logic to specify security policies. From now on, we shall only quantify over the domain *app*, so in the following we shall omit the sort annotation in the existential quantifier. The predicate *trans* is the recursive predicate defined in Equation (1) in the introduction. The constant *sink* denotes a service or resource that an unprivileged application tries to access via privilege escalation e.g. send SMS, or access to internet. The constant *contact* denotes the Contact provider app in Android. We also assume the following "static" predicates (i.e., their truth values do not vary over time):

- $system(x)$: x is a system app or process. By system app here we mean any app that is provided and certified by google (such as Google Maps, Google Play, etc) or an app that comes preinstalled to the phone.
- $hasPermissionToSink(y)$: y has permission to access the sink.
- $trusted(x)$: x is an app that the user trusts. This is not a feature of Android, rather, it is a specific feature of our implementation. We build into our implementation a 'trust' management app to allow the user a limited control over apps that he/she trusts.

The following policies refer to access patterns that are forbidden. So given a policy ϕ, the monitor at each state i make sure that $(\rho, i) \nvDash \phi$ holds. Assuming that $(\rho, i) \nvDash \phi$, where $i = |\rho|$, holds, then whenever a new event (i.e., the IPC call) e is registered at time t, the monitor checks that $(([\pi; e], [\tau; t]), i+1) \nvDash \phi$ holds. If it does, then the call is allowed to proceed. Otherwise, it will be terminated.

Table 1: Performance Table (ms)

Policy	Uncached	Cached
1	76.64	14.36
2	93.65	42.36
3	94.68	41.83
4	92.43	42.75
No Monitor	75.8	16.9

Table 2: Memory Overhead Table

Policy	Size(kB)	Overhead(%)
1	372	0.05
2	916	0.11
3	916	0.11
4	916	0.11

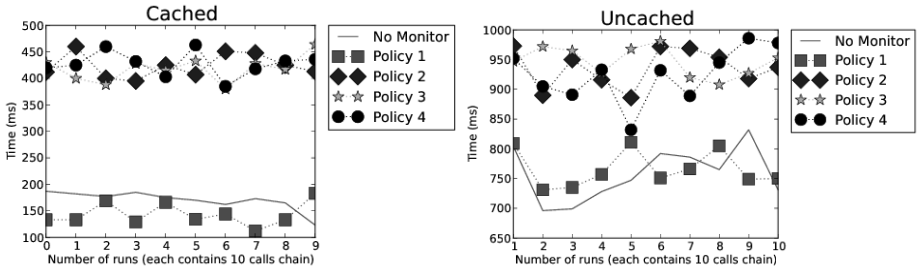
Note: on emulator with 49 apps and overall memory of around 800 mB

1. $\exists x.(call(x, sink) \wedge \neg system(x) \wedge \neg trusted(x))$.
   This is a simple policy where we block a direct call from any untrusted application to the sink. This policy can serve as a privilege manager where we dynamically revoke permission for application to access the sink regardless of the static permission it asked during installation.
2. $\exists_x(trans(x, sink) \wedge \neg system(x) \wedge \neg hasPermissionToSink(x))$.
   This policy says that transitive calls to a sink from non-system apps are forbidden, unless the source of the calls already has permission to the sink. This is then a simple privilege escalation detection (for non-system apps).
3. $\exists_x(trans(x, sink) \wedge \neg system(x) \wedge \neg trusted(x))$.
   This is further refinement to the policy in that we also give the user privilege to decide for themselves dynamically whether or not to trust an application. Untrusted apps can not make transitive call to the sink, but trusted apps are allowed, regardless of their permissions.
4. $\exists_x(trans(x, internet) \wedge \neg system(x) \wedge \neg trusted(x) \wedge \diamond(call(x, contact)))$.
   This policy allows privilege escalation by non-trusted apps as long as there is no potential for data leakage through the sink. That is, as soon as a non-system and untrusted app accesses contact, it will be barred from accessing the internet. Note that the use of non-metric operator $\diamond$ ensures that the information that a particular app has accessed contact is persistent. This policy resembles the well-known Chinese Wall policy [7] that is often used to manage conflict of interests. Here accessing contacts and connecting to the internet are considered as different conflict-of-interests classes.

5 Implementation

We have implemented the monitoring algorithm presented in the previous section in Android 4.1. Some modifications to the application framework and the underlying Linux kernel are neccessary to ensure our monitor can effectively monitor and stop unwanted behaviours. We have tested our implementation in both Android emulator and an actual device (Samsung Galaxy Nexus phone).

Our implementation consists of two parts: the codes that generate a monitor given a policy specification, and the modifications of Android framework and its Linux kernel to hook our monitor and to intercept IPCs and access to Android resources. The modification on Android framework mainly revolves around Activity Manager Service, a system component which deals with processing intent.



(a) (b)

Fig. 1: Timing of Calls

We add a hook in the framework to redirect permission checking (either starting activity, service, or broadcasting intent) to pass through our monitor first before going to the usual Android permission checking. The modification to the kernel consists mainly of additional system calls to interact with the framework, and a monitor stub which will be activated when the monitor module is loaded. To improve runtime performance, the monitor generation is done outside Android; it produces C codes that are then compiled into a kernel module, and inserted into Android boot image.

The monitor generator takes an input policy, encoded in an XML format extending that of RuleML. The monitor generator works by following the logic of the monitoring algorithm presented in Section 3. It takes a policy formula ϕ, and generates the required data structures and determines an ordering between elements of $Sub^*(\phi)$ as described earlier, and produces the codes illustrated in Algorithm 2, 3 and 1. The main body of our monitor lies in the Linux kernel space as a kernel module. The reason for this is that there are some cases where Android leaves the permission checking to the Linux kernel layer e.g., for opening network socket. However, to monitor the IPC events between Android components and apps, we need to place a hook inside the application framework. The IPC between apps is done through passing a data structure called *Intent*, which gets broken down into *parcels* before they are passed down to the kernel level to be delivered. So intercepting these parcels and reconstructing the original Intent object in the kernel space would be more difficult and error prone. The events generated by apps or components will be passed down to the monitor in the kernel, along with the application's user id. If the event is a call to the sink, then depending on the policy that is implemented in the monitor, it will decide to whether block or allow the call to proceed. We do this through our custom additional system calls to the Linux kernel which go to this monitor.

Our implementation places hooks in four services, namely accessing internet (opening network sockets), sending SMS, accessing location, and accessing contact database. For each of this sink, we add a virtual UID in the monitor and treat it as a component of Android. We currently track only IPC calls

through the Intent passing mechanism. This is obviously not enough to detect all possible communications between apps, e.g., those that are done through file systems, or side channels, such as vibration setting (e.g., as implemented in SoundComber [24]), so our implementation is currently more of a proof of concept. In the case of SoundComber, our monitor can actually intercepts the calls between colluding apps, due to the fact that the sender app broadcasts an intent to signal receiver app to start listenting for messages from the covert channels.

We have implemented some apps to test policies we mentioned in Section 4. In Table 1 and Figure 1, we provide some measurement of the timing of the calls between applications. The policy numbers in Table 1 refer to the policies in Section 4. To measure the average time for each IPC call, we construct a chain of ten apps, making successive calls between them, and measure the time needed for one end to reach the other. We measure two different average timings in miliseconds (ms) for different scenarios, based on whether the apps are in the background cache (i.e., suspended) or not. We also measure the time spent on the monitor actually processing the event, which is around 1 ms for policy 1, and around 10 ms for the other three policies. This shows that the time spent in processing the event is quite low, but more overhead comes from the space required to process the event (there is a big jump in overall timing from simple rules with at most 2 free variables to the one with 3 free variables). Figure 1 shows that the timing of calls over time for each policy are roughly the same. This backs our claim that even though our monitor implements history-based access control, its performance does not depend on the size of the history. Table 2 shows the memory footprints of the security monitors. The first column in the table shows the actual size of the memory required by each monitor, and the second column shows the percentage of the memory of each monitor relative to the overall available memory. As can be seen from the table, the memory overhead of the monitors is negligible.

6 Conclusion, related and future work

We have shown a policy language design based on MTL that can effectively describe various scenarios of privilege escalation in Android. Moreover, any policy written in our language can be effectively enforced. The key to the latter is the fact that our enforcement procedure is trace-length independent. We have also given a proof-of-concept implementation on actual Android devices and show that our implementation can effectively enforce RMTL policies.

We have already discussed related work in runtime monitoring based on LTL in the introduction. We now discuss briefly related work in Android security. There is a large body of works in this area, more than what can be reasonably surveyed here, so we shall focus on the most relevant ones to our work, i.e., those that deal with privilege escalation. For a more comprehensive survey on other security extensions or analysis, the interested reader can consult [8]. QUIRE [12] is an application centric approach to privilege escalation, done by tagging the intent objects with the caller's UID. Thus, the recipient application can check

the permission of the source of the call chain. IPC Inspection [15] is another application centric solution that works by reducing the privilege of the recipient application when it receives a communication from a less privileged application. XManDroid [8] is a system centric solution, just like ours. Its security monitor maintains a call graph between apps. It is the closest to our solution, except that we are using temporal logic to specify a policy, and our policy can be modified modularly. This way, a system administrator can have flexibility in designing a policy that is suited to the system in question. Moreover, should an attacker find a way to circumvent the current monitor, we can easily modify the monitor to enforce a different policy that addresses the security hole.

Our policy language is also more expressive as we can specify both temporal and metric properties. As a result, XManDroid will have better performance in general (exploiting the persistent link in the graph by using cache), yet there are policies that our monitor can enforce but XManDroid cannot. For example, consider Policy 4 in Section 4. XManDroid can only express whether an application has the permission to access contact, but not the fact that contact was accessed in the past. So in this case XManDroid would forbid an app with permission to access contact to connect to the internet, whereas in our case, we prevent the connection to the internet only after contact was actually accessed. TaintDroid [13] is another system-centric solution, but it is designed to track data flow, rather than control flow, via taint analysis, so privilege escalation can be inferred from leakage of data.

We currently do not deal with quantifiers directly in our algorithm. Such quantifiers are expanded into purely propositional connectives (when the domain is finite), which is exponential in the number of variables in the policy. As an immediate future work, we plan to investigate whether techniques using *spawning automata* [5] can be adapted to our setting to allow a "lazy" expansion of quantifiers as needed. It is not possible to design trace-length-independent monitoring algorithms in the unrestricted first-order LTL [5], so the challenge here is to find a suitable restriction that can be enforced efficiently.

Acknowledgment. This work is partly supported by the Australian Research Council Discovery Grant DP110103173.

References

1. Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. In *LICS*, pages 390–401. IEEE Computer Society, 1990.
2. David A. Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *FSTTCS*, volume 2 of *LIPIcs*, pages 49–60. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.
3. David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Algorithms for monitoring real-time properties. In *RV*, volume 7186 of *LNCS*, pages 260–275. Springer, 2012.
4. Andreas Bauer, Rajeev Gore, and Alwen Tiu. A first-order policy language for history-based transaction monitoring. In *Proc. 6th Intl. Colloq. Theoretical Aspects of Computing (ICTAC)*, volume 5684 of LNCS, pages 96–111. Springer, 2009.
5. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *RV*, volume 8174 of *LNCS*, pages 59–75, 2013.

6. Julian Bradfield and Colin Stirling. Modal mu-calculi. In *HANDBOOK OF MODAL LOGIC*, pages 721–756. Elsevier, 2007.
7. D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*. IEEE, 1989.
8. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS12*, 2012.
9. Patrick P. F. Chan, Lucas Chi Kwong Hui, and Siu-Ming Yiu. Droidchecker: analyzing android applications for capability leak. In *WISEC*, pages 125–136. ACM, 2012.
10. Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *ISC 2010*, volume 6531 of *LNCS*, pages 346–360, 2011.
11. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.
12. M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smartphone operating systems. In *20th USENIX Security Symposium*, 2011.
13. W. Enck, P. Gillbert, 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 *OSDI*, 2010.
14. William Enck, Machigar Ongtang, and Patrick Drew McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.
15. A. P. Felt, H. Wang, A. Moschuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, 2011.
16. Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1996.
17. Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS12*, 2012.
18. Hendra Gunadi and Alwen Tiu. Efficient runtime monitoring with metric temporal logic: A case study in the android operating system. *CoRR*, abs/1311.2362, 2013.
19. Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS*, volume 2280 of LNCS, pages 342–356. Springer, 2002.
20. Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In *Logic of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, 1985.
21. Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you're looking for. In *DefCon 18*, 2010.
22. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
23. Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Real-time runtime verification on chip. In *RV*, volume 7687 of *LNCS*, pages 110–125. Springer, 2013.
24. R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.
25. P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic specifications. In *Proc. of RV04*, 2004.
26. Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electr. Notes Theor. Comput. Sci.*, 113:145–162, 2005.