Trace-Length Independent Runtime Monitoring of Quantitative Policies in LTL

Xiaoning Du, Yang Liu and Alwen Tiu

School of Computer Engineering, Nanyang Technological University

Abstract. Linear temporal logic (LTL) has been widely used to specify runtime policies. Traditionally this use of LTL is to capture the qualitative aspects of the monitored systems, but recent developments in metric LTL and its extensions with aggregate operators allow some quantitative policies to be specified. Our interest in LTL-based policy languages is driven by applications in runtime Android malware detection, which requires the monitoring algorithm to be independent of the length of the system event traces so that its performance does not degrade as the traces grow. We propose a policy language based on a past-time variant of LTL, extended with an aggregate operator called the counting quantifier to specify a policy based on the number of times some sub-policies are satisfied in the past. We show that a broad class of policies, but not all policies, specified with our language can be monitored in a trace-length independent way without sacrificing completeness, and provide a concrete algorithm to do so. We implement and test our algorithm in an existing Android monitoring framework and show that our approach can effectively specify and enforce quantitative policies drawn from real-world Android malware studies.

1 Introduction

Linear temporal logic (LTL) has been widely used as a specification language to specify runtime properties of systems and languages. Traditionally, this use of LTL is concerned mainly with qualitative properties, such as relative ordering of events, or eventuality of events, etc. Our interest in the LTL-based policy languages is motivated by the demand for Android malware detection. In this setting, some attack patterns cannot be stated as pure LTL formulas as they require specifications of quantitative measures such as frequency of certain activities (e.g., sending SMS) commonly found in botnet attacks. Recent studies [22,21] indicate that Android malware is increasingly designed to turn infected phones into botnets, so to be practically useful, any monitoring framework for Android needs to take into account quantitative measures in their policy specifications.

One way to detect the kind of botnet attacks mentioned above is to count the number of certain events, such as SMS messages sent from an app, and notify the user once the count goes beyond some limit. To design a monitoring framework that can enfoce this kind of policies, one approach is to build into LTL a notion of counting of events [7], or more generally, aggregate operators [6]. The main problem is that monitoring algorithms for such extensions have not been well studied, and can be very inefficient, e.g., PSPACE complete (in the size of policy and the trace) for the extension of LTL with the counting quantifier [7], and PTIME (in the size of the trace) when the policy is fixed. In

the online monitoring of OS kernels, where near real-time decisions need to be made, the dependence of the monitor on the size of the trace would make it impractical even if its complexity is PTIME (assuming the policy is fixed), as its performance would degrade as the trace size grows. We attempt to address this problem in a minimal setting to demonstrate that it is possible to design a monitoring framework that is expressive enough to specify various quantitative properties and enforceable efficiently.

In this work, we propose an extension of Past Time LTL (PTLTL) [19], named PTLTL$_{cnt}$, to support the *counting quantifier*, which is motivated and extended from [7], and arithmetic functions and relations. PTLTL$_{cnt}$ considers only the fragment of PTLTL with past time operators, as this is sufficient for our purpose to enforce history-sensitive access control. For our intended application of monitoring Android applications, once we fix the policy to be monitored, the monitoring algorithm space requirement and runtime should be constant, i.e., independent of the length of the system event trace. Following [8], we call this type of monitoring algorithms as trace-length independent (TLI) monitoring algorithms. Note that we require that the generated TLI algorithms to be complete with respect to the policy specifications; otherwise the problem would be trivial as one could simply make various ad hoc restrictions such as restricting the time window for the monitoring. In [19], it is shown that a trace-length independent monitor can be generated for every formula of PTLTL. For richer logics, such as those considered in [7,8] and our own PTLTL$_{cnt}$, this is not always possible, i.e., there are formulas for which the monitor needs to store the entire history of events. For example, in PTLTL$_{cnt}$ one can write a formula that compares the numbers of two events, say e_1 and e_2. Let x and y denote the number of past occurrences of events of e_1 and e_2, respectively. To check the relation $x < y$ at any state, we would need to keep the counts of both e_1 and e_2; such counts would grow as the trace grows, so the space requirement for monitoring this formula is not bounded. We could only store $|x - y|$ in this case, but this absolute value can still grow infinitely.

As far as we know, there has been so far no study on trace-length independence monitoring for LTL with aggregate operators like the counting quantifier. To solve this issue, we first formally identify the precise characteristics of the class arithmetic relations that can be monitored in a trace-length independent way. Then we show that if all arithmetic relations in a PTLTL$_{cnt}$ formula are TLI-monitorable, the formula itself is also TLI-monitorable. More importantly, we show how to construct a TLI monitoring algorithm when all relations are TLI-monitorable.

We have performed a number of case studies on Android to show the practicality of our specification language for malware detection. We have implemented the proposed language and algorithm based on an existing Android monitoring framework called LogicDroid [17]. The experimental results shows that our approach can effectively specify and enforce a range of quantitative policies drawn from real-world Android malware.

Organization Section 2 presents the formal syntax and semantics of PTLTL$_{cnt}$. Section 3 proposes the trace-length independent monitoring algorithm for PTLTL$_{cnt}$ with univariate countingparts. This is generalized this to the multivariate case in Section 4. Some Android policy examples are introduced in Section 5. Section 6 describes the implementation of our algorithms for monitoring in LogicDroid. The related works are

discussed in Section 7. Section 8 concludes the paper. Due to space constraints, some proofs are omitted but they will be made available in an extended version of this paper.

2 The Policy Specification Language PTLTL$_{cnt}$

In this section, we formally introduce PTLTL$_{cnt}$ as an extension of PTLTL extended with a *counting quantifier*, which counts how many times a sub-policy has been satisfied in the past, as well as arithmetic operators and relations. Our counting quantifier has a slightly different semantics compared to that of [7] as explained later. Our language admits the usual arithmetic operators such as $+$, $-$ and $\times$, and relations $=$, $<$, $\leq$ and $\geq$. We assume a countably infinite set of constants. We use a, b, c and d to range over constant symbols of type integers. We denote with AP the set of propositional variables. Elements of AP are ranged over by P, Q and S. We assume an infinite set $\mathcal{V}$ of variables of type integers, whose elements are ranged over by x, y and z. Terms are built from constants, variables and arithmetic operators, and are denoted by s, t, u and v.

The syntax of PTLTL$_{cnt}$ is defined via the following grammar:

$$\phi := \bot \mid AP \mid (t > 0) \mid \neg\phi \mid \phi \vee \phi \mid \bullet\phi \mid \phi \mathbb{S} \phi \mid \mathcal{C}x : \langle \phi, \phi \rangle.\phi$$

The operators are those of PTLTL except for the counting quantifier $\mathcal{C}$ and the relation $t > 0$. The variable x in $\mathcal{C}x : \langle \phi_1, \phi_2 \rangle.\varphi$ is a bound variable, whose scope is over φ, so x is not free in either ϕ_1 or ϕ_2. Intuitively, the meaning of $\mathcal{C}x : \langle \phi_1, \phi_2 \rangle.\varphi$ is as follows: suppose that ϕ_2 is true at exactly n states since the latest state where ϕ_1 holds; then the instance of φ with x mapped to n must also be true. The formula ϕ_1 acts as a *counter reset* condition. We assume the reader is familiar with the notion of free and bound variables. We assume that bound variables in a formula are pairwise distinct. We write $\phi(x_1, \ldots, x_n)$ to mean that the free variables of ϕ are in $\{x_1, \ldots, x_n\}$ and we write $\phi(t_1, \ldots, t_n)$ to denote the instance of $\phi(x_1, \ldots, x_n)$ where t_i is substituted for x_i.

In the definition of formulas, we have kept a minimum number of logical operators. The omitted operators can be derived using the given operators, e.g., propositional operators such as $\top$ (truth), $\wedge$ (conjunction), $\rightarrow$ (implication), and modal operators such as $\blacklozenge$ (sometime in the past), which is defined as $\blacklozenge\phi \equiv \top \mathbb{S} \phi$, and $\blacksquare$ (globally in the past), which is defined as $\blacksquare\phi \equiv \neg\blacklozenge\neg\phi$. Note also that all other arithmetic relations can be derived from the relation of the form $(t > 0)$ and logical connectives: $s > t \equiv (s - t) > 0$, $s \leq t \equiv \neg(s > t)$, $s = t \equiv (s \leq t) \wedge (t \leq s)$, $s \geq t \equiv s > t \vee s = t$, and $s < t \equiv s \leq t \wedge \neg(s = t)$.

The semantics of PTLTL$_{cnt}$ is defined with respect to a finite trace model, as in [19]. A trace is just a sequence of states, where each state itself consists of a set of atomic propositions. These atomic propositions correspond to events of interests that are being monitored in a system. We assume an interpretation function I which maps constant symbols to integers, and arithmetic operators and relation symbols to their corresponding semantic counterparts. We assume the usual arithmetic operators, and in addition, depending on applications, we may assume a fixed set of function symbols denoting computable functions over the integer domain. Since terms and relations can contain variables, we additionally need to interpret these variables. This is done via a *valuation function*, i.e., a function from variables to integers. Formally, given an interpretation function I and a valuation function ν, the interpretation of a term t, written $t^{I,\nu}$ is

defined as in first-order logic [16]. However, since we shall only work within a fixed interpretation, we shall drop the superscript I in the following semantics definition.

A *model* for PTLTL$_{cnt}$ is a triple (ρ, ν, i), where ρ is a trace, ν is a valuation function and i is a natural number. For a trace ρ, we write ρ_i to denote its i-th *state*. For a valuation ν, we write $\nu[x \mapsto n]$ to denote the function which is identical to ν except for the valuation of x, i.e., $\nu[x \mapsto n](y) = \nu(y)$, when $y \neq x$, and $\nu[x \mapsto n](x) = n$. The satisfiability relation between a model (ρ, ν, i) and a formula ϕ, written $\rho, \nu, i \vDash \phi$, is defined by induction on ϕ below, where $\rho, \nu, i \nvDash \phi$ if $\rho, \nu, i \vDash \phi$ is false.

- $\rho, \nu, i \nvDash \phi$ if $i < 1$ or $i > |\rho|$.
- $\rho, \nu, i \nvDash \bot$.
- $\rho, \nu, i \vDash P$ iff $P \in \rho_i$.
- $\rho, \nu, i \vDash t > 0$ iff $t^\nu > 0$ is true.
- $\rho, \nu, i \vDash \neg\phi$ iff $\rho, \nu, i \nvDash \phi$.
- $\rho, \nu, i \vDash \phi \vee \psi$ iff $\rho, \nu, i \vDash \phi$ or $\rho, \nu, i \vDash \psi$.
- $\rho, \nu, i \vDash \bullet\phi$ iff $i > 1$ and $\rho, \nu, i-1 \vDash \phi$.
- $\rho, \nu, i \vDash \phi_1 \mathbb{S} \phi_2$ iff $\rho, \nu, i \vDash \phi_2$, or $\rho, \nu, i \vDash \phi_1$ and $\rho, \nu, i-1 \vDash \phi_1 \mathbb{S} \phi_2$ with $i > 1$.
- $\rho, \nu, i \vDash \mathcal{C}x : \langle \phi, \psi \rangle.\varphi$ iff $\rho, \nu[x \mapsto n], i \vDash \varphi$ where
  $n = |\{j | r \leq j \leq i \text{ and } \rho, \nu, j \vDash \psi\}|$ and $r = max(\{j | \rho, \nu, j \vDash \phi, j \leq i\} \cup \{1\})$

We write $\rho, i \vDash \phi$ when $\rho, \nu, i \vDash \phi$ for every valuation ν.

Example 1. For an authentication server (e.g., bank) which validates a user's credential, a common login policy can be that if a user fails to enter the correct password three times in a row, then the user's account is temporarily disabled. Let us consider only two system events: a correct password was entered (cp), and a wrong password was entered (wp) by a particular user. The logic policy can be specified as follows:

$$\blacksquare\,[\neg(cp \wedge wp) \wedge (\mathcal{C}x : \langle cp, wp \rangle.x < 3)]. \tag{1}$$

The first conjunct expresses a consistency property, i.e., a password entered cannot be both correct and wrong at the same time. The variable x stores the number of times a wrong password was entered since the last time a correct password was entered (or since the beginning of the trace, if no correct password has been entered so far). Consider the event trace $\rho = [\{wp\}; \{cp\}; \{wp\}; \{wp\}; \{cp\}; \{wp\}]$. Then formula (1) above is true at every state. □

In general, the counting quantifier can be used to express quantitative properties within a 'session' (e.g., an authentication session, a life cycle of a process, etc). One could introduce two events: $start$ and end, to mark the beginning and the end of a session. Then to check that the number of occurrences of an event e within a session is less than n, for example, one can simply use the formula $\mathcal{C}x : \langle start, e \wedge \neg end \rangle.\ x < n$ in conjunction with other formulas expressing the well-formedness of a session (e.g., every end corresponds to a $start$, etc). If e is a simple event (e.g., the wp event in Example 1), one could encode this in LTL using standard temporal operators, but at the expense of conciseness, i.e., one needs to expand the parameter n into n instances of $e \wedge \neg end$. For example, Example 1 can be alternatively specified as

$$\blacksquare[\neg(cp \wedge wp) \wedge \neg(\bullet wp \wedge \bullet(wp \wedge \bullet wp)))].$$

That is, there cannot be three consecutive wp events any time in the past. However, this is the case only when there are no events being monitored other than cp and wp. When other events are possible, then we need to specify that events other than cp can happen in between two consecutive wp events. In general, in a formula $\mathcal{C}x : \langle \phi_1, \phi_2 \rangle . \phi_3$, any of the ϕ_i could be a complicated temporal formula, e.g., it could involve nested counting quantifiers and other temporal operators. In such a case, the encoding into pure LTL becomes less obvious and less concise. We shall see more examples drawn from Android malware study in Section 5.

Our counting quantifier is a generalization of Bauer et. al.'s counting quantifier [7]. Their counting quantifier is semantically defined as follows: $\rho, \nu, i \vDash \mathcal{N}x : \psi.\varphi$ iff $\rho, \nu[x \mapsto n], i \vDash \varphi$ where $n = |\{j | 1 \leq j \leq i \text{ and } \rho, \nu, j \vDash \psi\}|$. However, in terms of expressiveness, they are actually equivalent, as shown next.

Proposition 1. *The counting quantifiers $\mathcal{C}$ and $\mathcal{N}$ are equivalent, i.e., one can be defined in terms of the other.*

Proof. (*Outline.*) The quantifier $\mathcal{N}$ can be encoded using $\mathcal{C}$ as follows: $\mathcal{N}x : \phi.\psi \equiv \mathcal{C}x : \langle \bot, \psi \rangle . \varphi$. Conversely, $\mathcal{C}$ can be encoded using $\mathcal{N}$ as follows:

$$\mathcal{C}x : \langle \phi, \psi \rangle . \varphi \;\equiv\; \mathcal{N}z : \phi.\, \mathcal{N}x : (\psi \wedge \mathcal{N}y : \phi.y = z).\, \varphi.$$

Note that the subformula $\mathcal{N}y : \phi.y = z$ acts essentially as a counter reset. It is not difficult to check from their semantics that these encodings are correct. □

Note that although $\mathcal{C}$ can be encoded using $\mathcal{N}$, the encoding introduces nested occurrences of $\mathcal{N}$ and one needs to compare at least two counting variables. In general, policies involving two or more counting variables are impossible to enforce in a trace-length independent way, as our example in the introduction shows. We could have simply used the original counting quantifier $\mathcal{N}$, but we would then have to use the encoding above, that involves comparing two or more variables, to capture the idea of a session. Such encodings would thus obscure the underlying structure of the problem, and makes it harder to systematically generate TLI monitors from a given specification. For instance, the policy described in Example 1 uses only one counting variable when expressed using $\mathcal{C}$, and results in Section 3 would guarantee the existence of TLI monitors for that particular policy. Had we chosen to encode it using $\mathcal{N}$, we would have to work harder in order to show that the policy is in fact TLI monitorable.

3 Trace-length Independent Monitoring for PTLTL$_{cnt}$

In a setting with limited storage and computation resource (e.g., an OS kernal or embedded devices), an online monitoring algorithm that requires the storage of the entire event trace is no practical, even if its complexity is PTIME. Ad hoc restrictions such as limiting the time window or enforcing bounded storage of events are not desirable as they may introduce incompleteness with respect to the policies being enforced, i.e., there may be violations to the policies that can only be detected on a trace of events longer than what could fit in the storage. In early work such as [19], monitoring algorithms are designed so that their memory requirement is constant, when one fixes the

formula to be monitored, without compromising the completeness of the monitor with respect to the formula. For example, for PTLTL [19], one needs to maintain only two states of each subformula of a policy to enforce without losing completeness of the algorithms. Following [8], we call this type of monitoring algorithms as trace-length independent monitoring algorithms, and we call formulas that can be monitored in a trace-length independent way trace-length independent formulas, or *TLI-formulas* for short, and such formulas are said to be *TLI-monitorable*.

Since a main difference between PTLTL$_{cnt}$ and PTLTL is the presence of arithmetic relations, we first look at a class of relations $\phi(x_1, \ldots, x_n)$ that are TLI-monitorable (the precise definition will be given later). If all relations in a formula are all TLI-monitorable, then it is straightforward to check that the formula itself must be TLI-monitorable. In this section, we look at the univariate case, i.e., functions with arity 1. We generalize this to the multivariate case in Section 4.

In the following, all variables range over integers and the domains of functions are assumed to be tuples of integers, unless otherwise stated. Further, given a function F of arity n, we denote with $\varphi_F(x_1, \ldots, x_n)$ the relation $F(x_1, \ldots, x_n) > 0$.

Definition 1. *Given a function F, we construct a binary function F_G as follows:*

$$F_G(x_1, \ldots, x_n) = \begin{cases} 0 & \text{if } F(x_1, \ldots, x_n) \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

We call F_G the *G-function* of F, which can be seen as the characteristic function of φ_F.

Definition 2. *A function F defined on domain $\mathbb{D}$ is said to be* periodic *over interval I with period T if we have*

$$F(x) = F(x + T)$$

for all values of $x \in \mathbb{D} \cap I$, with also $(x + T) \in \mathbb{D} \cap I$.

Definition 3. *A total function $F : \mathbb{N} \to \mathbb{R}$ is said to be* lower-bounded periodic*, or* lb-periodic *for short, if there is a $b \in \mathbb{N}$ such that F is periodic on interval $[b, +\infty)$.*

Definition 4. *Let $F : \mathbb{N} \to \mathbb{R}$ be a total function. Then φ_F is* TLI-monitorable *if there are two constants c and k, with $c \geq k \geq 1$ and $c, k \in \mathbb{N}$, such that*

$$\varphi_F(x) = H(\varphi_F(x-1), \ldots, \varphi_F(x-k))$$

for $x \geq c$, where H is a total computable Boolean function.

Intuitively, TLI-monitorable relations are those for which the $F(x) > 0$ can be solved incrementally, i.e., if we know the truth values of $F(y) > 0$ for a finite number of $y < x$, we would be able to compute the truth value of $F(x) > 0$. Notice that there is no need to store the actual value of the counting variable x nor $F(x)$ in this incremental computation; all that matters is the truth value of the relation $F(x) > 0$. Thus the space required for monitoring such relations remain constant irrespective of the value of x.

Example 2. Let $F(x) = x^2 - 8x + 15$, where $x \in \mathbb{N}$. The G-function of F in this case is

$$F_G(x) = \begin{cases} 0 & \text{if } 3 \leq x \leq 5 \\ 1 & \text{otherwise} \end{cases}$$

This is because $F(x) \leq 0$ is satisfied only when $3 \leq x \leq 5$. □

We now characterize precisely the class of relations which are TLI-monitorable according to Definition 4.

Lemma 1. *Let $F : \mathbb{N} \to \mathbb{R}$ be a total function. Then φ_F is TLI-monitorable if F_G is lb-periodic.*

Example 3. The function F_G in Example 2 is lb-periodic, with the lower bound 6, and period 1. So φ_F is TLI-monitorable according to Lemma 1. □

We now prove the converse: every TLI-monitorable function must be lb-periodic.

Lemma 2. *Given a total function $F : \mathbb{N} \to \mathbb{R}$, if φ_F is TLI-monitorable, then F_G is lb-periodic.*

With sufficiency and necessity proved in Lemma 1 and Lemma 2 respectively, we get the following theorem:

Theorem 1. *Given a total function $F : \mathbb{N} \to \mathbb{R}$, φ_F is TLI-monitorable iff F_G is lb-periodic.*

The abstract characterization in Theorem 1 is in a way quite obvious from the definition of TLI-monitorable relations. The important part is that monitorability is associated with periodic "relations" (F_G) rather than functions (F). That is, F may not be periodic yet still be TLI-monitorable (see Theorem 2 below). In concrete applications, since we are usually only given the function F and not F_G, the difficult problem is in deciding whether F_G is lb-periodic given F. In the following, we show some broad classes of functions for which their G-functions are lb-periodic.

Theorem 2. *Given a computable total function $F : \mathbb{N} \to \mathbb{R}$, φ_F is TLI-monitorable if F satisfies one of the following conditions:*

1. *F is lb-periodic.*
2. *F is monotonously increasing/decreasing.*
3. *F is a univariate polynomial.*

Now, we shall look at the monitoring problems for formulas in which all relations and functions are univariate and TLI-monitorable.

To monitor a formula $\mathcal{C}x : \langle \phi, \psi \rangle . \varphi(x)$, we first extract all functions involving x from φ. Suppose as $F(x)$ is one of the functions extracted. According to Theorem 1, the G-function $F_G(x)$ for $F(x)$ should be lb-periodic for $\varphi_F(x)$ to be TLI-monitorable. Then the track of $\varphi_F(x)$ can be seen as a path ended with a loop. Then the values of $\varphi_F(x)$ repeats periodically as x increases. We can thus quotient the values of x based on the period of F_G to form a finite set of equivalence classes.

Definition 5. *Given a TLI-monitorable relation φ_F with lb-periodic function F_G that is periodic over $[b, +\infty)$ with period T, we define the* equivalence class *for the domain of F as:*

$$[i] = \begin{cases} \{i\} & \text{if } i < b \\ \{a | ((a-b) \bmod T + b) \equiv i\} & \text{otherwise} \end{cases}$$

Obviously, the number of the equivalence classes should be $(b + T)$ and every element within an equivalence class will have the same φ_F value. To check the value of $\varphi_F(x)$ at any point x, it is sufficient to check the value of φ_F at the equivalence class of x. For simplicity, each equivalence class is indexed with the minimal element in the set. The reset condition ϕ is orthogonal to the issue of quotienting the values of x; when it is satisfied, the count for x is reset to 0. We write $\mathfrak{r}(\rho, i, \mathcal{C}x : \langle\phi, \psi\rangle.\varphi)$ to denote the index of equivalence class to which the counter of ψ belongs with model (ρ, ν, i).

In the following, we shall assume that, given a formula φ, every subformula of φ of the form $\mathcal{C}x : \langle\phi, \psi\rangle.\varphi$ has the property that x occurs exactly once in a univariate function. This is not a real limitation as the case where x is vacuous or occurs more than once (in different univariate functions) can be encoded into an equivalent formula where each quantified variable occurs exactly once. If x is vacuous in φ, then $\mathcal{C}x : \langle\phi, \psi\rangle.\varphi$ is logically equivalent ot φ. If x occurs twice, i.e., φ is, e.g., $\varphi_1(x) \wedge \varphi_2(x)$, then we rewrite the formula to an equivalent one: $\mathcal{C}x : \langle\phi, \psi\rangle.\mathcal{C}y : \langle\phi, \psi\rangle.\varphi_1(x) \wedge \varphi_2(y)$. This holds because x and y are bound to the same value at every state. The same technique generalizes to the cases where x occurs more than twice in φ.

Given the above restriction on the syntax of formulas, in a formula $\mathcal{C}x : \langle\phi, \psi\rangle.\varphi$, we can extract exactly one function F where x is used. As in the case of monitoring algorithms in [19,17], the key to get the trace-length independence property is to express the semantics of all logical operators in a recursive form, i.e., the truth value of φ at state i is a function of truth values of subformulas of φ and/or the truth value of φ at state $i - 1$. All operators except the counting quantifier are already in recursive form. The next theorem shows that the semantics of the counting quantifier also admits a recursive form, when all relations in the formula to be monitored are univariate and TLI-monitorable.

Theorem 3. *Given a model (ρ, ν, i) and a closed formula $\mathcal{C}x : \langle\phi, \psi\rangle.\varphi(x)$ where x occurs in a function F, and $\varphi_F(x)$ is TLI-monitorable with the lb-periodic function F_G that is periodic over $[b, +\infty)$ with period T, the following holds for every $1 < i \leq |\rho|$:*

$$
\begin{aligned}
&\rho, \nu, i \vDash \mathcal{C}x : \langle\phi, \psi\rangle.\varphi \text{ iff } \rho, \nu, i \vDash \phi, \text{ and } \rho, \nu, i \vDash \varphi(0);\\
&\text{or } \rho, \nu, i \vDash \psi, \mathfrak{r}(\rho, i-1, \mathcal{C}x : \langle\phi, \psi\rangle.\varphi) < b, \text{ and } \rho, \nu, i \vDash \varphi_F((\mathfrak{r}(\rho, i-1, \mathcal{C}x : \langle\phi, \psi\rangle.\varphi) + 1);\\
&\text{or } \rho, \nu, i \vDash \psi, \mathfrak{r}(\rho, i-1, \mathcal{C}x : \langle\phi, \psi\rangle.\varphi) \geq b, \text{ and } \rho, \nu, i \vDash \varphi_F((\mathfrak{r}(\rho, i-1, \mathcal{C}x : \langle\phi, \psi\rangle.\varphi) + 1 - b)\\
&\qquad \bmod T + b);\\
&\text{or } \rho, \nu, i \vDash \neg\phi, \rho, i \vDash \neg\psi, \text{ and } \rho, \nu, i-1 \vDash \mathcal{C}x : \langle\phi, \psi\rangle.\varphi(x)
\end{aligned}
$$

Once we get the semantics of all logical operators of PTLTL$_{cnt}$ in a recursive form, we can use dynamic programming to design a trace-length independence algorithm for PTLTL$_{cnt}$. Following the algorithm for PTLTL [19], we compute the truth values of every subformula of a given formula φ, at exactly two successive states. However, for quantified formulas, its subformulas would contain free variables, and their truth values would thus depend on the values of x. To avoid coding valuation of variables explicitly in the monitoring algorithm, we need to instantiate x to concrete terms before computing their truth values. Given the restriction imposed on the formulas as discussed above, we can associate each quantified variable x in φ with exactly one univariate function;

```
struct CntInfo {
  int indexEC;
  int period;
  int lowerBound;
}
```

Fig. 1: CntInfo Data Structure

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

```
1 Init(ρ, φ, prev, cur, cnt);
2 for j = 1 to i do
3     Iter(ρ, j, φ, prev, cur, cnt);
4 return cur[idx(φ)];
```

let us call it F_x. Since φ_{F_x} is TLI-monitorable, by Definition 5, we can compute a finite set of equivalence classes for the values of x. Suppose this set has n elements $\{e_1, \ldots, e_n\}$. Then we need to instantiate x only with these n values. So given a subformula $\mathcal{C}x : \langle\phi, \psi\rangle.\theta(x)$ of φ, we define its immediate subformulas as: ϕ, ψ and $\varphi(e_i)$, for $1 \leq i \leq n$. We let $Sub(\varphi)$ denote the set of all subformulas of φ.

Now we will describe how monitoring can be done for ϕ, given ρ and $1 \leq i \leq |\rho|$. Let $\phi_1, \phi_2, \ldots, \phi_m$ be an enumeration of $Sub(\phi)$ respecting the order that any formula has an enumeration number greater than that of all its subformulae. Following the notations in [17], we can assign to each $\psi \in Sub(\phi)$ an index i, such that $\psi = \phi_i$ 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 $\rho, \nu, i-1 \vDash \phi_k$ and the truth value of $cur[k]$ corresponds to the truth value of the judgment $\rho, \nu, i \vDash \phi_k$.

Recall that each quantified variable is used in exactly one univariate function. For each variable x, we keep a data structure *CntInfo*, shown in Figure 1, which stores the lower bound (*lowerBound*) and the period of (*period*) the G-function, and the index of the equivalence class induced by the G-function (*indexEC*). The initialization of an instance of *CntInfo* is conducted in *init_counter()*, which will set *lowerBound* and *period* the accordingly, and zero the *indexEC*. The array $cnt[1, \ldots, l]$ in both *Init* and *Iter* algorithm stores a list of *CntInfo* objects associated with each variables. We assign an index $idx_c(x)$ to each variable x, and $cnt[idx_c(x)]$ maintains the information associated with the counter variable x.

The main monitoring algorithm (Algorithm 1) is divided into two sub-procedures: the initialisation procedure (Algorithm 2) and the iterative procedure (Algorithm 3). In the pseudocode of the algorithms, we overload some logical symbols to denote operators on boolean values. It is straightforward to see that, once the formula to be monitored is fixed, the space required to run the algorithm does not grow with the length of traces. In particular, the values of the counter variables (the *indexEC* field) is bounded, i.e., it never grows beyond $period + lowerBound$.

4 Extension to Multivariate Relations

We now look at the case where relations can be multivariate. We shall restrict our discussions to the bivariate case; the extension to the multivariate case is straightforward and does not require any new techniques so we omit details here.

Algorithm 2: $Init(\rho, \phi, prev, cur, cnt)$

```
1  for k = 1 to m do
2      switch φ_k do
3          case ⊥ cur[k] ← false;
4          case P cur[k] ← P ∈ ρ_1;
5          case ¬ψ cur[k] ← ¬cur[idx(ψ)];
6          case t > 0 cur[k] ← t^ν > 0;
7          case ψ_1 ∨ ψ_2 cur[k] ← cur[idx(ψ_1)] ∨ cur[idx(ψ_2)];
8          case •ψ cur[k] ← false;
9          case ψ_1 S ψ_2 cur[k] ← cur[idx(ψ_2)];
10         case Cx : ⟨ψ_1, ψ_2⟩.φ(x)
11             init_counter(cnt[idx_c(x)]);
12             if !cur[idx(ψ_1)] then
13                 if cur[idx(ψ_2)] then
14                     cnt[idx_c(x)].count++;
15                     cur[k] ← cur[idx(φ(1))];
16             else cur[k] ← cur[idx(φ(0))];
17 return cur[idx(φ)];
```

Definition 6. *Let $F : \mathbb{N}\times\mathbb{N} \to \mathbb{R}$ be a total function, then φ_F is TLI-monitorable if there are constants c_1, c_2 and k_1, k_2, with $c_1 \geq k_1 \geq 1$, $c_2 \geq k_2 \geq 1$ and $c_1, c_2, k_1, k_2 \in \mathbb{N}$, such that*

$$\varphi_F(x,y) = \begin{cases} F(x,y) > 0 & \text{if } x < c_1 \text{ and } y < c_2, \\ H\begin{pmatrix} \varphi_F(x-k_1,y), & \ldots, & \varphi_F(x-1,y), \\ \varphi_F(x-k_1,y-1), & \ldots, & \varphi_F(x,y-1), \\ \varphi_F(x-k_1,y-k_2), & \ldots, & \varphi_F(x,y-k_2) \end{pmatrix} & \text{otherwise.} \end{cases}$$

where H is a total computable Boolean function.

Theorem 4. *Given a total function $F : \mathbb{N}\times\mathbb{N} \to \mathbb{R}$, if there are constants T_x, c_x, T_y and c_y such that $F_G(x,c)$ is lb-periodic with period T_x for any $c \in \mathbb{N} \geq c_y$, and $F_G(d,y)$ is lb-periodic with period T_y for any $d \in \mathbb{N} \geq c_x$, then φ_F is TLI-monitorable.*

Essentially, Theorem 4 says that φ is TLI-monitorable if the period of a projection of F into one of its parameter is independent of the other parameter, once the value of that parameter exceeds a certain threshold. This allows us to quotation the values of each parameters into their own equivalence classes independently of each other.

The monitoring algorithm is surprisingly the same as the univariate case. We still need to adopt the same restriction regarding the occurrences of variables as in the univariate case, i.e., that each quantified variable appears exactly once in a bivariate function. The main difference between the univarite and the bivariate case is finding the right lower bound and the periods of each variables, a process which takes place outside the algorithm; once these parameters are defined, the monitoring algorithm proceeds as in the univariate case.

Algorithm 3: $Iter(\rho, i, \phi, prev, cur, cnt)$

```
1  prev ← cur;
2  for k = 1 to m do
3      switch φ_k do
4          case ⊥ cur[k] ← false;
5          case P cur[k] ← P ∈ ρ_i;
6          case ¬ψ cur[k] ← ¬cur[idx(ψ)];
7          case t > 0 cur[k] ← t^ν > 0;
8          case ψ1 ∨ ψ2 cur[k] ← cur[idx(ψ1)] ∨ cur[idx(ψ2)];
9          case •ψ cur[k] ← prev[idx(ψ)];
10         case ψ1 𝕊 ψ2 cur[k] ← cur[idx(ψ2)] ∨ (cur[idx(ψ1)] ∧ prev[idx(ψ2)]);
11         case 𝒞x : ⟨ψ1, ψ2⟩.φ(x)
12             if !(cur[idx(ψ1)] ∨ cur[idx(ψ2)]) then cur[k] ← pre[k];
13             else
14                 if cur[idx(ψ1)] then
15                     cnt[idx_c(x)].indexEC ← 0;
16                 else
17                     if cur[idx(ψ2)] then
18                         n ← ++cnt[idx_c(x)].indexEC;
19                         lowerBound ← cnt[idx_c(x)].lowerBound;
20                         period ← cnt[idx_c(x)].period;
21                         if n ≥ lowerBound + period then
22                             cnt[idx_c(x)].indexEC ←
                               (n − lowerBound) mod period + lowerBound;
23                 cur[k] ← cur[idx(φ(cnt[idx_c(x)].indexEC))];
24 return cur[idx(φ)];
```

The extension to the multivariate case follows the same idea, i.e., a sufficient condition for φ_F, when F is an n-ary function, to be monitorable is that the period of any of its projection is independent of the other projections.

5 Case Studies in Android

In this section, some concrete policies in Android systems are provided as case studies for PTLTL$_{cnt}$. In the rest of this paper, we assume the following atomic propositions in Android OS. S_i (or E_i) means the application with UID i starts to run (or stops running). M_i means the application with UID i sends out a message. I_i means the application with UID i opens an Internet connection socket. F_i : the application with UID i forks a new child process. C_i means the application with UID i accesses the contact database.

The following policies refer to the malicious access patterns that are forbidden in Android systems. At any moment, if $\rho, \nu, |\rho| \not\models \phi$ holds, and when a new event P occurs, the monitor checks whether $[\rho; P], \nu, |\rho| + 1 \not\models \phi$ holds. If it does, the process forwards

fluently. Otherwise, a suspicious alert will send to the user. As for the specific limit on specific counted amount, we just give a rough estimation for illustration purpose.

1. $\mathcal{C}x : \langle S_i, M_i \wedge \neg E_i \rangle.(x > 5)$
   This policy is to guarantee that an app with UID i cannot send more that 5 SMS messages during a single run, which is inspired by [3] for stopping unintended SMS transmissions. In [21], authors found that current Android botnets are exploiting SMS messages to gather money by sending SMS to premium-rate numbers. With the specified policy, it helps to make possible discovery of an Android bot.
2. $\mathcal{C}x : \langle S_i, I_i \wedge \neg E_i \rangle.(x > 200)$
   This policy says that an app cannot open Internet connection socket for more than 200 times in a single run. If an Android app aims at flooding a targeted server to launch a DDoS attack, one way to achieve this is to open massive Internet connections. This policy can help to control the amount of Internet connections, thus preventing some potential malware.
3. $\mathcal{C}x : \langle S_i, F_i \wedge \neg E_i \rangle.(x > 2^{16})$
   This policy says that during the life cycle of an application, it is not allowed to create more than 2^{16} child processes to exhaust the pid, i.e., the process identifier in Linux kernel. *RageAgainstTheCage* [1] is a well-known exploit in Android, which can perform unauthorized privileged actions by gaining the root access. This malware uses a vulnerability in Android kernel to get the root privilege by keeping forking the child process to 2^{16}. With this policy specified using PTLTL$_{cnt}$, constraint is set to how many child processed an application can fork in a single run, it will be much helpful to prevent this attack.

6 Implementation and Evaluation

We have implemented the monitoring algorithm for PTLTL$_{cnt}$ and evaluated it on LogicDroid platform [17], which is a modified Android system based on Android 4.1. The Android IPC (Inter-process communication) calls, like opening Internet socket, sending SMS and accessing contact database hooked by LogicDroid form the set of events against which policies in PTLTL$_{cnt}$ need to be checked. Since LogicDroid does not yet implement hooks to detect the start or end of a process or an app, the reset conditions in our example policies are not applicable. In particular, policy 3 in Section 5, which counts the child processes forked by an app, has not yet been tested due to the lack of support for process forking detection in LogicDroid.

The list of six policies adopted in our experiments is presented in Figure 2. among which the first two are that introduced in Section 5, and the others are artificial examples to evaluate the robustness of our approach. To keep the diversity of the policies, the number of counters in the six policies is 1, 1, 2, 3, 6, 10 separately. Also there are policies with complicate reset conditions. Each policy is implemented as a Linux kernel module according to the algorithm described in Section 3. For every counter variable in the tested policies, the initialization of fields *period* and *lowerBound* in *CntInfo* struct are currently done manually.

To test the practicability and efficiency of our approach, we implement a fuzzy testing app to trigger three kinds of IPC calls (M_i, I_i and C_i in Section 5) randomly.

1. $\mathcal{C}x : \langle S_i, M_i \wedge \neg E_i \rangle.(x > 5)$
2. $\mathcal{C}x : \langle S_i, I_i \wedge \neg E_i \rangle.(x > 200)$
3. $\mathcal{C}x : \langle \bot, M_i \rangle.\mathcal{C}y : \langle \bot, I_i \rangle.K(x, y) > 0$ with the definition of function K as follows:

$$K(x,y) = \begin{cases} 3x - 4y & \text{if } x < 5 \text{ and } y < 4, \\ K(x-3, y) & \text{if } x \geq 5 \text{ and } y < 4, \\ K(x, y-2) & \text{if } x < 5 \text{ and } y \geq 4, \\ K(x-3, y-2) & \text{otherwise.} \end{cases}$$

   Note that each projection of function K becomes periodic once $x > 9$ and $y > 13$, so it is easy to show that φ_K is TLI-monitorable.
4. $(I_i \mathbb{S} C_i) \wedge \mathcal{C}x : \langle \bot, M_i \rangle.\mathcal{C}y : \langle \bot, I_i \rangle.\mathcal{C}z : \langle \bot, C_j \rangle.H(x, y, z) > 0$ with function H defined as follows:

$$H(x,y,z) = \begin{cases} 3x - 4y + 2z & \text{if } x < 19 \text{ and } y < 13 \text{ and } z < 5, \\ H(x-9, y, z) & \text{if } x \geq 19 \text{ and } y < 13 \text{ and } z < 5, \\ H(x, y-2, z) & \text{if } x < 19 \text{ and } y \geq 13 \text{ and } z < 5, \\ H(x, y, z-1) & \text{if } x < 19 \text{ and } y < 13 \text{ and } z \geq 5, \\ H(x-9, y-2, z) & \text{if } x \geq 19 \text{ and } y \geq 13 \text{ and } z < 5, \\ H(x, y-2, z-1) & \text{if } x < 19 \text{ and } y \geq 13 \text{ and } z \geq 5, \\ H(x-9, y, z-1) & \text{if } x \geq 19 \text{ and } y < 13 \text{ and } z \geq 5, \\ H(x-9, y-2, z-1) & \text{otherwise.} \end{cases}$$

   Each projection of function H is periodic when $x > 19$, $y > 13$ and $z > 5$, so φ_H is TLI-monitorable.
5. $\mathcal{C}x_1 : \langle \bot, \bullet C_i \rangle \wedge \mathcal{C}x_2 : \langle \bot, \neg I_i \rangle \wedge \mathcal{C}x_3 : \langle I_i, C_i \rangle \wedge \mathcal{C}x_4 : \langle \bot, \bullet I_i \rangle \wedge \mathcal{C}x_5 : \langle \bot, M_i \rangle \wedge \mathcal{C}x_6 : \langle \bot, I_i \rangle$
6. $\mathcal{C}x_1 : \langle \bot, \bullet C_i \rangle \wedge \mathcal{C}x_2 : \langle \bot, \neg I_i \rangle \wedge \mathcal{C}x_3 : \langle I_i, C_i \rangle \wedge \mathcal{C}x_4 : \langle \bot, \bullet I_i \rangle \wedge \mathcal{C}x_5 : \langle \bot, M_i \rangle \wedge \mathcal{C}x_6 : \langle \bot, I_i \rangle \wedge \mathcal{C}x_7 : \langle \bot, \neg M_i \rangle \wedge \mathcal{C}x_8 : \langle \bot, \bullet M_i \rangle \wedge \mathcal{C}x_{10} : \langle \mathcal{C}x_9 : \langle \bot, \neg C_i \rangle, M_i \rangle$

Fig. 2: Additional policies used in the experiments.

For the evaluation, we measure the detection time of the monitoring process (i.e., the execution time of a policy monitoring kernel module for processing a single event) and the memory used by the system with the extra policy monitoring kernel module. All the experiments are conducted in the LogicDroid emulator on 64-bit Ubuntu 14.04LTS with 16GB RAM and an Intel Xeon(R) CPU E5-1650 v2 with 3.50GHz. Our implementation and the models shown in this section are available in [2].

Figure 3 shows the time used by the monitor for a single event check. To measure the detection time, we launch the fuzzy testing app to send 1000 IPC calls continuously, therefore the monitor kernel module will be invoked 1000 times. We record the detection time in every 50 calls as shown in Figure 3. It can be seen from the figure that the time used for each policy monitoring kernel module is stable, i.e., does not increase with trace length grows. There is no obvious difference between the time cost for different policies, and the average checking time is 6 to 8 microseconds. Figure 4 shows the memory usage of the emulator with the six different kernel modules. To consider the

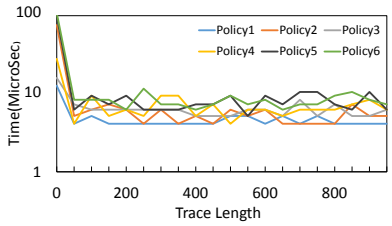

Fig. 3: Time of Single Round Checking

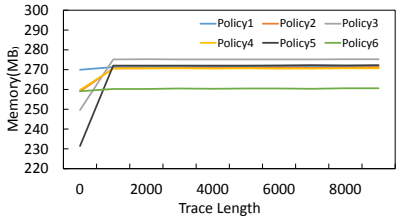

Fig. 4: Memory Usage

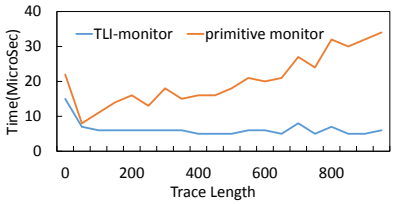

Fig. 5: Comparison of Checking Time

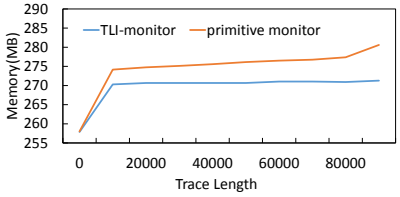

Fig. 6: Comparison of Memory Usage

impact of the additional kernel module on memory usage in the emulator more accurately, we measure the memory when every 1000 IPC calls are triggered for continuous 10 times. Clearly, all of the memory usage measured for the emulator with different kernel module installed in turn are quite stable (i.e., does not increase with the time), which supports our claim that the proposed algorithm is trace-length independent.

Note that the six policies module tested in the experiment are with a increasing number of counters. From the results shown in Figure 3 and Figure 4, we can know that the number of counters involved in a policy has little timing and memory influence.

To give an emperical validation of the effectiveness of our monitoring algorithm, comparison experiments have been done with a direct primitive counting mechanism, where all events will be recorded and the entire history will be searched to get the statistics of count when the monitor checks the validation. The results of monitoring the policy 3 in Figure 2 are shown in Figure 5 and Figure 6. For the detection time, it is following the previous measure experiment. While the memory is measured when every 10000 IPC calls are triggered for continuous 10 times. We choose a different setting to make visible the gradually increasing tendency of memory used by the primitive monitor. As can be seen, there is obvious increase of the time and space required by the primitive monitor as the trace length grows.

7 Related Work

This section lists some recent works on counting quantifier combined with different kinds of logic theories, aiming to increase the expressiveness of the logic. A brief outline of the history of trace-length independent runtime monitoring will also be given.

Inspired by the aggregation operators in database query language like SQL, Basin *et al.* [6] extend metric first-order temporal logic (MFOTL) with aggregation operators, like SUM, CNT, MAX and AVG, and proposed a monitoring algorithm for language. The core of this work is to translate policies specified with the extended MFOTL to the corresponding extended relational algebra. For their monitoring algorithm, functions are handled similarly to Prolog. Even through some optimizations are taken to accelerate computations in monitoring, the aggregation operators are out of their consideration. Another language, SOLOIST [11], is based on a many-sorted first-order metric temporal logic and extended with new temporal modalities that support aggregate operators for events occurring in a certain time window. For its monitoring, Bianculli *et al.* [12] proposed to translate the formulae in SOLOIST to formulae of CLTLB($\mathscr{D}$) [10], and Bersani *et al.* [9] presented an approach to encode SOLOIST formulae into QF-EUFIDL formulae. Nevertheless, both approaches depend on SMT-solver to do the final satisfiability checking. The evaluations of the above two works show that increasing time and memory will be needed when the length of the trace grows.

Laroussinie *et al.* [20] presented a quantitative extension for LTL, called CLTL, allowing to specify the number of states satisfying certain sub-formulas along paths, which provided the same semantics with ours. They also showed even though CLTL formulae can be translated into classical LTL, an exponential blow-up in formula size is inevitable. As for the satisfiability and model-checking problems for CLTL, they turned out to be EXPSPACE-complete, but PSPACE-complete when restricting CLTL to a fragment. Actually this fragment is just a subset of the TLI-formulas defined in our work, for which the relation function will grow monotonously with any involved count increasing.

Other monitoring approaches that provide support for different kinds of aggregations are LarvaSat [13], LOLA [14], as well as rule-based EAGLE [4], RULER [5] and LOGFIRE [18], and one based on algebraic alternating automata [15], However, all monitoring algorithms for the above languages still need to record the specific counted values, even though most of them avoided storing the entire trace history. In principle, these counters can increase indefinitely, so their space complexity is not constant unlike our monitoring algorithms.

There are some works [19,8,17] concentrate on designing trace-length independent monitoring algorithms. In particular, this work can be seen as an effort to extend the LogicDroid framework to incorporate the counting quantifier of [7]. Although the concept of trace-length independence is proposed in 2013, there are also some prior works which imply this property in their algorithm design. For the best of our knowledge, this is first work on designing trace-length independence algorithms involving counting quantifier, not even other aggregation operators. For now, we are the first one to implement a trace-length independent runtime verification algorithm for the logic language with a counting quantifier.

8 Conclusion

We have presented a formal policy specification language PTLTL$_{cnt}$ that allows expressions of quantitative policies. We consider the questions of when a formula is trace-length independent monitorable. For univariate relations, we obtain sufficient and necessary conditions for the relations to be TLI-monitorable. We then discussed an extension to the multivariate relations. Assuming the relations are all TLI-monitorable, we construct a TLI monitoring algorithm for PTLTL$_{cnt}$. We have implemented and tested our monitoring algorithm, and the experimental results more or less confirm our theoretical results. Currently, we have not yet addressed the integration of the counting quantifier with metric operators and recursive predicates of [17]. This is a subject of immediate future work. We also plan to look to incorporate other, more expressive aggregrate operators from [6].

Acknowledgment We thank the anonymous referees for their helpful comments. This research is supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

References

1. Rageagainstthecage. `http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage`, 2011.
2. Tli monitoring of ltl extended with a counting quantifier. `http://pat.sce.ntu.edu.sg/xndu/fm2015`, 2015.
3. S. Arzt, K. Falzon, A. Follner, S. Rasthofer, E. Bodden, and V. Stolz. How useful are existing monitoring languages for securing android apps? In *ATPS*, pages 107–122, 2013.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
5. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from eagle to ruler. *Journal of Logic and Computation*, 20(3):675–706, 2010.
6. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. In *RV*, pages 40–58, 2013.
7. A. Bauer, R. Goré, and A. Tiu. A first-order policy language for history-based transaction monitoring. In *ICTAC*, pages 96–111. 2009.
8. A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *RV*, pages 59–75, 2013.
9. M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. San Pietro. Smt-based checking of soloist over sparse traces. In *FASE*, pages 276–290. 2014.
10. M. M. Bersani, A. Frigeri, A. Morzenti, M. Pradella, M. Rossi, and P. S. Pietro. Constraint ltl satisfiability checking without automata. *Journal of Applied Logic*, 12(4):522 – 557, 2014.
11. D. Bianculli, C. Ghezzi, and P. San Pietro. The tale of soloist: a specification language for service compositions interactions. In *Formal Aspects of Component Software*, pages 55–72. 2013.
12. D. Bianculli, S. Krstic, C. Ghezzi, and P. San Pietro. From soloist to cltlb (d): Checking quantitative properties of service-based applications. 2013.

13. C. Colombo, A. Gauci, and G. J. Pace. Larvastat: Monitoring of statistical properties. In *RV*, pages 480–484, 2010.
14. B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *TIME*, pages 166–174, 2005.
15. B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma. Collecting statistics over runtime executions. *Form. Methods Syst. Des.*, 27(3):253–274, 2005.
16. M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1996.
17. H. Gunadi and A. Tiu. Efficient runtime monitoring with metric temporal logic: A case study in the android operating system. In *FM*, pages 296–311. 2014.
18. K. Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):1–28, 2012.
19. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *TACAS*, pages 342–356, 2002.
20. F. Laroussinie, A. Meyer, and E. Petonnet. Counting ltl. In *TIME*, pages 51–58, 2010.
21. H. Pieterse and M. S. Olivier. Android botnets on the rise: Trends and characteristics. In *ISSA2*, pages 1–5, 2012.
22. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *SP*, pages 95–109, 2012.