2018 ACM/IEEE 40th International Conference on Software Engineering

A Temporal Permission Analysis and Enforcement Framework for Android

Alireza Sadeghi
Department of Informatics
University of California, Irvine, USA
alirezs1@uci.edu

Reyhaneh Jabbarvand
Department of Informatics
University of California, Irvine, USA
jabbarvr@uci.edu

Negar Ghorbani
Department of Informatics
University of California, Irvine, USA
negargh@uci.edu

Hamid Bagheri
Department of Computer Science and Engineering
University of Nebraska, Lincoln, USA
bagheri@unl.edu

Sam Malek
Department of Informatics
University of California, Irvine, USA
malek@uci.edu

ABSTRACT

Permission-induced attacks, i.e., security breaches enabled by permission misuse, are among the most critical and frequent issues threatening the security of Android devices. By ignoring the *temporal* aspects of an attack during the analysis and enforcement, the state-of-the-art approaches aimed at protecting the users against such attacks are prone to have low-coverage in detection and high-disruption in prevention of permission-induced attacks. To address this shortcomings, we present TERMINATOR, a temporal permission analysis and enforcement framework for Android. Leveraging temporal logic model checking,TERMINATOR's *analyzer* identifies permission-induced threats with respect to dynamic permission states of the apps. At runtime, TERMINATOR's *enforcer* selectively leases (i.e., temporarily grants) permissions to apps when the system is in a safe state, and revokes the permissions when the system moves to an unsafe state realizing the identified threats. The results of our experiments, conducted over thousands of apps, indicate that TERMINATOR is able to provide an effective, yet non-disruptive defense against permission-induced attacks. We also show that our approach, which does not require modification to the Android framework or apps' implementation logic, is highly reliable and widely applicable.

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**; **Access control**; • **Theory of computation** → Modal and temporal logics;

KEYWORDS

Android, Access Control (Permission), Temporal Logic

ACM Reference format:
Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek. 2018. A Temporal Permission Analysis and Enforcement Framework for Android. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 12 pages.
https://doi.org/10.1145/3180155.3180172

1 INTRODUCTION

Popular mobile operating systems, such as Android, apply a permission-based model to patrol resources that each application is allowed to access. In this model, critical system and application resources are protected by an explicit permission, which then must be obtained by any application that would like to access the resources. Yet, in the past few years since the inception of Android, a number of flaws have been identified in its permission mechanism that can lead to serious security and privacy breaches [26]. A large body of research, thus, has been devoted to address detection and prevention of permission-induced attacks in Android [45, 50].

The state-of-the-art approaches, however, fail to consider the temporal aspects of permission-induced attacks during the analysis and enforcement, thereby suffer from shortcomings that aggravate their effectiveness. Detection of several permission-induced attacks, such as those exploiting the *TOCTOU* (Time of Check to Time of Use) vulnerability in Android [28, 48], requires careful consideration of the *order* of events. Hence, existing *detection* techniques, which ignore the element of time in their analysis, are prone to miss important security breaches. Additionally, due to the highly dynamic state of an Android system, the identified security vulnerabilities may only be exploitable at specific time intervals, e.g., when some specific permissions are granted. Hence, the existing conservative *prevention* techniques, which regardless of the system state enforce security rules *permanently*, tend to produce plenty of false alarms. As a result, users can be unnecessarily disrupted, even in the absence of material security threats, and prevented from taking full advantage of the apps on their device.

Finally, the proposed approaches are mostly realized through modification of either the Android framework [17, 29, 31, 43, 52] or the implemenation logic of apps [9, 19, 44, 53]. But, such modifications are not necessarily expected, nor properly tested by the application developers, resulting in all sorts of undesirable side effects, such as app crashes and unexpected behaviors. To address this state of affairs, a pragmatic approach for detection and prevention should explicitly consider the temporal aspects of attack during analysis and enforcement. Moreover, the realization of the approach should be naturally compatible with the implementation practices in Android.

This paper contributes a novel approach and accompanying tool suite, called TERMINATOR, short for **T**emporal P**ermi**ssio**n** **A**nalysis and enforcemen**t** framew**o**rk for And**r**oid. Unlike all prior techniques, TERMINATOR incorporates the notion of time as a first class entity in both detection and prevention of permission-induced attacks. Our approach has the potential to greatly improve our ability to thwart permission-induced attacks by introducing the concept of *temporal permissions*, i.e., the temporary granting of permissions to apps. Specifically, constructed atop *temporal logic*, TERMINATOR leverages temporal permissions to (1) formulate dynamic aspects of the system over time and reason about the security properties thereof as the system transitions from one state to another (risk detection), and (2) regulate app permissions at runtime based on the current state of the system (risk prevention).

TERMINATOR provides a safe, reliable, yet non-disruptive approach to protect mobile users against permission misuses. Upon receiving a permission request from an app, TERMINATOR evaluates the security posture of the system with respect to the current state of the granted-permission configuration as well as potential threats conservatively identified via the state-of-the-art static analysis tools. If granting the requested permission does not lead to a real security threat given the current state of the system, TERMINATOR *leases* (i.e., temporarily grants) that permission to the requester. The leased permission is then automatically revoked as soon as a change in the system status is observed that may lead to realization of an identified security threat. TERMINATOR uses TLA+ model checker (TLC) [54] as an analysis engine for temporal permissions. To prevent permission-induced attacks, TERMINATOR relies on the Android's dynamic permission mechanism without needing to make any modification to the Android framework or the implementation logic of apps.

Our experiments indicate that TERMINATOR is up to 68% more successful in preventing permission-induced attacks, while issuing significantly less (56-100%) false alarms. It also causes less disruption in the availability of permission-protected app functionality due to restrictive permission configurations.

To summarize, this paper makes the following contributions:

- *Theory*: To the best of our knowledge, this is the first attempt at leveraging temporal logic and incorporating the notion of time in modeling and analyzing the security properties of Android;
- *Tool*: A fully automated framework, TERMINATOR, that realizes the idea of temporal permissions for Android, which we have made publicly available [4];
- *Experiments*: Empirical evaluation of the approach on real-world Android apps demonstrating its efficacy.

The remainder of this paper is organized as follows. Section 2 motivates our research through various examples of permission-induced security attacks. Section 3 formally specifies those attacks and introduces our approach to effectively thwart them. Section 4 provides details of our approach and its implementation. Section 5 presents the experimental evaluation of the research. The paper concludes with an outline of the related research and future work.

2 PERMISSION-INDUCED ATTACKS

To motivate the research and demonstrate the need for temporal permissions, we describe four types of permission-induced security attacks in Android, identified in prior research [26]. Permission-induced attacks are security breaches enabled by Android permissions misuse. This section elaborates on the attack scenarios summarized in Figure 1. We will later show how temporal permissions help thwart these attack scenarios with minimum disruption.

2.1 Privilege Escalation

Privilege escalation occurs when an application with less privilege is not restricted from accessing components of a more privileged application [16]. In the case of the particular example shown in Figure 1(a), Mal App can indirectly reach the permission-protected interface of the Privileged App, by exploiting the vulnerability of the Victim App — that is, an unprotected exposed interface, shown to be quite common in the app markets [20]. The *collusion attack* [46], carried out by multiple malicious apps through combining a set of permissions to perform unauthorized actions, is also categorized under this group of attacks.

The state-of-the-art techniques for preventing inter-app security attacks [45] conservatively assume that this vulnerability is exploitable, as soon as the apps are installed on the device. However, a more careful look at the timeline of the attack scenario, shown in Figure 1(a), would reveal that the presented security vulnerability is only exploitable during the "unsafe" time slot, where the following two conditions hold at the same time: (1) the malware and victim apps are both active, i.e., running in foreground or background, and (2) permission P is granted to *Victim App*. If those applications are installed but not active, the vulnerability cannot be exploited. On the other hand, if permission P is not granted to the victim app, the permission-protected interface of the other app is not accessible.

2.2 Unsafe PendingIntent

In Android, *PendingIntent* is a wrapper around Intent that enables performing the Intent's action in future, even if the original app that sent the Intent is not active anymore. For this purpose, Android transfers the permission and identity (UID) of the sender app to the target app that receives the PendingIntent. As such, careless use of PendingIntent can lead to severe security consequences. Examples include the privilege leakage vulnerability in the Android Settings application (CVE-2014-8609) [1].

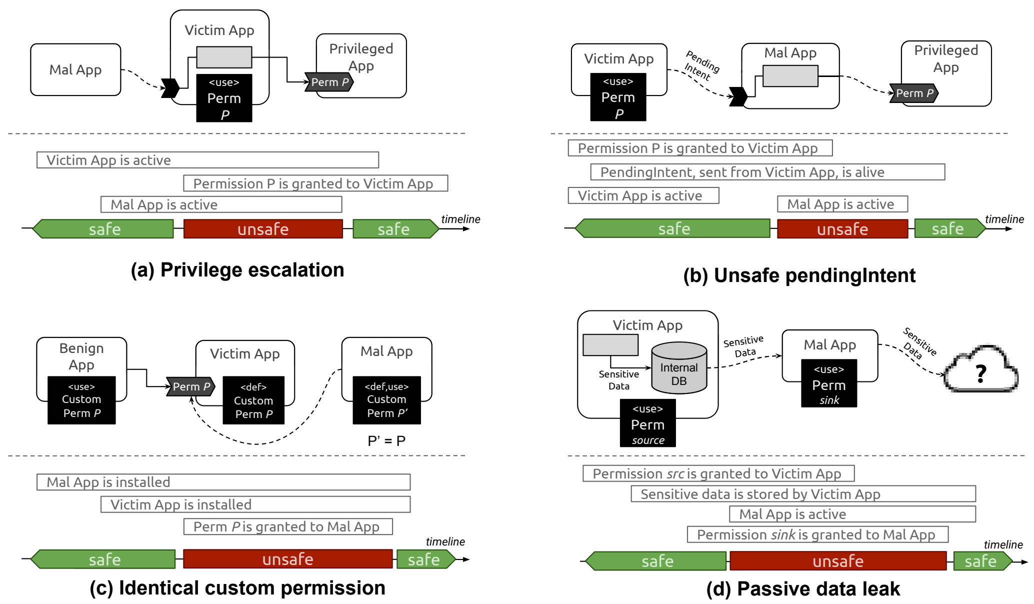
For this reason, Android's developer guidelines strongly discourage using blank base PendingIntents: *"the base Intent you supply should have the component name explicitly set to one of your own components, to ensure it is ultimately sent there and nowhere else* [3]." Despite that, many app developers fail to follow such security principle in action.

Figure 1(b) shows an example of using unsafe PendingIntent, exploited by *Mal App* to illegally access permission-protected interface provided by *Privileged App*. This example is similar to the privilege escalation attack, illustrated in Figure 1(a), except that the conditions for exploitability are more relaxed in two ways: first, the *Victim App* does not need to be necessarily active, and second, its permission P may be revoked prior to malware executing the wrapped Intent.

2.3 Identical Custom Permission

Besides the predefined built-in permissions, such as `SMS`, `LOCATION`, etc., Android apps can define their own custom permissions and request those permissions from other apps. However, the custom permission model suffers from a security vulnerability rooted in a design flaw: *"If two apps define the same custom permission, whichever app is installed first is the one whose definition is used"* [10].

A malicious app can exploit the custom permission vulnerability to illegally access the interface of another app, protected by that custom permission. A sample attack scenario is shown in Figure 1(c). In this example, *Victim* and *Mal* apps have both defined the same custom permission, i.e., the names of permissions P and P', defined by the `<permission>` element in the manifest are identical. Since the malicious app is installed prior to the victim app, permission P', defined in the manifest of the *Mal App* at the Normal level, is the one recognized by the Android framework. Consequently, *Mal App* can access the interface defined by the victim app, which is intended



(a) Privilege escalation

(b) Unsafe pendingIntent

(c) Identical custom permission

(d) Passive data leak

Figure 1: Examples of permission-induced security attacks that could be thwarted using temporal permission. In each example, the top of the figure shows the elements involved in the attack, and the bottom shows a possible attack scenario over time. The permission can be leased, i.e., temporarily granted, during the "safe" time slots. In all of the scenarios, the malicious communications are distinguished by dashed lines.

to be only accessible to those requesting the custom permission *P*, such as *Benign App*. The custom permission breach can happen even though the permissions with the same name have different protection levels.[1] Essentially, the malware can define a permission with `normal` protection level, rather than `dangerous` or `signature` protection level, to evade user attention and interaction.[2]

2.4 Passive Data Leak

A passive data leak occurs when an app does not properly protect its internal database that contains sensitive data [56]. A malware can exploit this vulnerability by retrieving the stored data, without having the permission needed for directly accessing such sensitive information. Thereafter, the malicious app can transfer the sensitive data to an untrustworthy location.

Figure 1(d) depicts an example of passive data leakage. In this attack scenario, *Victim App* with an access to *Sensitive Data* due to obtaining *source* permission (e.g., permission for accessing phone identifier), stores this information in its internal database, which is not properly protected. As a result, *Mal App* can retrieve the sensitive data and send it to an untrustworthy location, if it has been granted with a *sink* permission (e.g., SMS permission).

From the attack scenarios shown in Figure 1, we can see that the notion of time is critical in the precise description of all attacks. In other words, a precise analysis should keep track of the security posture of the system as it moves from one state to another over time. Hence, to formally describe the attack scenarios we need to formulate the system properties in terms of time. For this purpose, we leverage temporal logic, as described in the next section.

[1]The protection level indicates the trustworthiness of an application that may be granted this permission.
[2]The system automatically grants *normal* permission to a requesting application at installation, without asking for the user's explicit approval.

3 TEMPORAL PERMISSION

In this section, we describe a formal model of the Android system with a focus on its security properties such as permission status. Using this model, we then define a set of safety formulas corresponding to the permission-induced security attacks, described in the previous section. Finally, we demonstrate that control of the permissions granted to apps is sufficient for effectively thwarting all such attacks.

3.1 Modeling the Android System

We model the Android system as a *Kripke Structure*, a variation of *Transition System* that mathematically models dynamic systems [22].

```
/** Determines whether app is installed on the device */
Installed(app)
/** Determines whether app is running, either in the foreground or
       background */
Active(app)
/** Determines whether permission is declared by app */
Declared(app, permission)
/** Determines whether permission is requested by app */
Requested(app, permission)
/** Determines whether permission is (requested by and) granted to
       app at runtime */
Granted(app, permission)
/** Determines whether app defines an exposed interface reaches to
       permission-required capability */
Exposed(app, permission)
/** Determines whether app sends a pendingIntent with blank base
      Intent (w/o explicit target) containing permission-required
      data payload */
BlankPI(app, permission)
/** Determines whether app retrieves permission-protected data (e.
      g., IMEI, location, etc)*/
Retrieve(app, permission, data)
/** Determines whether app sends data through permission-protected
       channels (e.g., SMS, Internet, etc.) */
Send(app, permission, data)
/** Determines whether data is stored by the app in an unprotected
       database*/
StoreUnprotected(app, data)
```

Figure 2: Atomic Propositions (AP) defined for modeling the security properties of Android system.

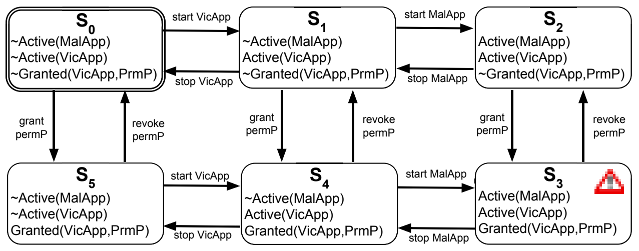


Figure 3: A subset of Kripke structure for a hypothetical Android system

Nodes represent the reachable states of the system, and edges represent state transitions. Each node is also labeled with a set of properties that hold in the corresponding state.

More formally, we model the system as a 4-tuple $M = (S, I, R, L)$, where S is a set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a total transition relation, and L is the labeling function that assigns to each state the subset of properties that are valid in the state. To define the labeling function, we first need to define *atomic propositions*, or *AP*, which is a set of boolean expressions that specify the properties of the system S. For instance, $Granted(app_a, perm_p)$ states that the permission p is granted to the app a. We use atomic propositions to define the labeling function as follows: $L : S \times AP \rightarrow \{true, false\}$. In other words, for each system state $s \in S$, the labeling function, L, determines whether the atomic proposition $ap \in AP$ holds at that state or not.

Modeling the Android operating system in its entirety with all its compound structures would be infeasible. We thus concentrate on the parts that are particularly relevant to the permission mechanism—how permissions are granted and maintained, and how they constrain the behavior of an application. Figure 2 provides the set of atomic propositions, defined as parameterized predicates. For instance, $Granted(app, permission)$ has two arguments, the first one is an *app* and the second one is a *permission* requested by that app.

As a concrete example, Figure 3 shows a small subset of the Kripke structure of a hypothetical Android system with six states: the initial state $s_0 \in I$ along with five other states, $s_{1-5} \in S$. Also, two kinds of actions triggering state transitions $r \in R$, are shown here, namely *start/stop* actions that alter the system configuration apropos of the *Active* proposition and *grant/revoke* actions that alter the system configuration apropos of the *Granted* proposition.

Interacting with the user and other environmental actors, Android system moves from one state to another in the Kripke structure over time. As a result, under specific sequence of actions, the system can move to an *unsafe* state—a state that violates the security of the system. For instance, in Figure 3, S_3 represents an unsafe state, corresponding to the red time slot of the privilege escalation attack scenario shown in Figure 1(a). S_3 is unsafe since a privilege escalation attack is possible when the system moves to this state.

3.2 Formulating Safety Rules

We specify *safety rules* in terms of conditions that need to hold throughout the states of the system. For this purpose, we leverage *linear-time temporal logic (LTL)*. Temporal logics enable specifying a system's behavior as it evolves over time. Indeed, in temporal logics, the truth of a statement is not fixed in the semantics, rather relies on the point in time when it is considered. Temporal logics, thus, besides the usual logical operators, such as *and*, *or*, *not*, and *implies*, also contain temporal operators, such as *eventually*, *always*, and *until*. For instance, "SMS permission can *eventually* be granted to the Messenger app", or "SMS permission should not be granted to the Messenger app *until* the Malware app is terminated", are two examples of such statements that can be expressed using temporal logic. In LTL, time is represented by a sequence of discrete time steps.

Privilege Escalation: The first formula, $SafetyRule_{PE}$, specifies the conditions needed to hold in order to prevent the privilege escalation attack (Recall Section 2.1).

$$\begin{aligned} &SafetyRule_{PE} \models \\ &\forall\, app_{vic}, app_{mal} \in Apps,\ p \in Permissions: \\ &Vul_{PE}(app_{vic}, app_{mal}, p) \Rightarrow \\ &\Box\neg(Granted(app_{vic}, p) \wedge Active(app_{mal})) \end{aligned} \tag{1}$$

The precondition of rule 1 checks for the privilege escalation vulnerability (Vul_{PE}), which is formulated as follows:

$$\begin{aligned} &Vul_{PE}(app1, app2, perm) := \\ &Requested(app1, perm)\ \wedge \neg Requested(app2, perm) \\ &\wedge Exposed(app1, perm) \end{aligned}$$

According to the above expression, for the given two apps, $app1$ and $app2$, and the Android permission $perm$, the hosting Android device is vulnerable to the privilege escalation attack, if $app1$, granted permission $perm$, exposes an unprotected interface to a capability protected by $perm$, while $perm$ is not requested by the other app.

$SafetyRule_{PE}$ states that the system is safe against the privilege escalation attack, if none of the system's apps expose the aforementioned vulnerability, or otherwise the unsafe permission of the vulnerable app should remain as "not granted" as long asmalicious app is active. Note the usage of temporal operator $\Box$, read *henceforth*[3], in the safety rule specified in formula 1, which states that the conditional consequent should hold in all future states.

Unsafe PendingIntent: The second formula, $SafetyRule_{UPI}$, specifies the conditions needed to hold in order to prevent the attacks exploiting an unsafe PendingIntent (Recall Section 2.2).

$$\begin{aligned} &SafetyRule_{UPI} \models \\ &\forall\, app_{vic}, app_{mal} \in Apps,\ p \in Permissions: \\ &Vul_{UPI}(app_{vic}, app_{mal}, p) \wedge Granted(app_{vic}, p) \Rightarrow \\ &\neg\Diamond Active(app_{mal}) \end{aligned} \tag{2}$$

The precondition of rule 2 checks for the unsafe PendingIntent vulnerability (Vul_{UPI}), which is formulated as follows:

$$\begin{aligned} &Vul_{UPI}(app1, app2, perm) := \\ &Requested(app1, perm)\ \wedge \neg Requested(app2, perm) \\ &\wedge\ BlankPI(app1, perm) \end{aligned}$$

Unlike the privilege escalation exploits, PendingIntent exploits do not require the breached permission to be granted to the vulnerable app prior to the attack. This is essentially because the required permission is already transferred to the mal app through the PendingIntent. Hence, the temporal operator $\Diamond$, read *eventually*[4], is used in the conditional consequent. According to formula 2, the system is safe against exploiting unsafe pendingIntent, if there is no such vulnerability or the vulnerable app is not granted with the breached permission. Otherwise, the system is unsafe as soon as the mal app is activated.

Identical Custom Permission: The third formula, $SafetyRule_{ICP}$, specifies the conditions that need to hold to

[3] $\Box\ \Phi$ means Φ is true at all future states
[4] $\Diamond\ \Phi$ means Φ is true in some future state

prevent the attacks exploiting the identical custom permission vulnerability (Recall Section 2.3).

$$\begin{aligned}&SafetyRule_{ICP} \models \\ &\forall app_{vic}, app_{mal} \in Apps,\ p \in Permissions: \\ &Vul_{ICP}(app_{vic}, app_{mal}, p) \wedge Installed(app_{mal}) \Rightarrow \\ &\square\neg(Granted(app_{mal}, p) \wedge Installed(app_{vic}))\end{aligned} \tag{3}$$

The precondition of rule 3 checks for the unsafe identical custom permission vulnerability (Vul_{ICP}), which is formulated as follows:

$$\begin{aligned}&Vul_{ICP}(app1, app2, perm) := \\ &Declared(app1, perm) \wedge Declared(app2, perm)\end{aligned}$$

Recall from Section 2.3 that the order of installation matters in the case of identical custom permission. To formulate this chronological order, *henceforth* temporal operator ($\square$) is used. According to rule 3, if (a potentially malicious) application with a declared custom permission p has been already installed on the device, no other app declaring the same permission is allowed to be installed, as long as that permission is granted to the first app.

Passive Data Leak: The last formula, $SafetyRule_{PDL}$ specifies the conditions needed to hold in order to prevent the leakage of sensitive data stored in an unprotected app database (Recall Section 2.4).

$$\begin{aligned}&SafetyRule_{PDL} \models \\ &\forall\ app_{vic}, app_{mal} \in Apps, p_1, p_2 \in Permissions: \\ &Vul_{PDL}(app_{vic}, app_{mal}, p_1, p_2) \wedge Granted(app_{vic}, p_1) \Rightarrow \\ &\neg\Diamond Granted(app_{mal}, p_2)\end{aligned} \tag{4}$$

Passive data leak vulnerability, formally defined below (Vul_{PDL}), occurs when a sensitive (i.e. permission-protected) data is sent out of the device by another app, via a (typically) permission-protected channel:

$$\begin{aligned}&Vul_{PDL}(app_{src}, app_{snk}, p_{src}, p_{snk}) := \exists\ data \in PhoneData: \\ &Requested(app_{src}, p_{src})\ \wedge Retrieve(app_{src}, p_{src}, data) \\ &\wedge StoreUnprotected(app_{src}, data) \\ &\wedge Requested(app_{snk}, p_{snk}) \wedge Send(app_{snk}, p_{snk}, data)\end{aligned}$$

According to rule 4, the system is safe against the passive data leak, if either there is not such a vulnerability or the vulnerable app has never been granted the permission to access sensitive data. Otherwise, the system is unsafe as soon as the malicious app is granted the permission, allowing the app to send data out of the device.

3.3 Leasing Temporal Permissions

To keep the Android device safe against the attack scenarios described in Section 2, one should guarantee that the corresponding safety rules hold at all times. A careful revisit of the safety rules (Rules 1–4) reveals that the $\neg$*Granted(app, permission)* proposition is incorporated in all formulas.[5] Therefore, permanently revoking specific permissions can guarantee the safety of the system. This approach, however, is too conservative as it revokes app permission even when the other criteria needed for exploitation of security vulnerability is not satisfied. In other words, since $\neg$*Granted* proposition is qualified in terms of time, it is not necessary to satisfy it over all system states. Instead, the app permission should only be revoked during specific unsafe states, and can be granted in the rest of system states.

[5]In the safety rule 2, $Vul_{UPI}() \wedge Granted() \Rightarrow \neg\Diamond Active()$ is logically equivalent to $\neg Vul_{UPI}() \vee \neg Granted() \vee \neg\Diamond Active()$.

Based on this intuition, we propose a defense mechanism against permission-induced attacks, called TERMINATOR. Upon receiving a permission request from an app, TERMINATOR *leases* (i.e., temporarily grants) that permission to the requester, only if granting the requested permission does not violate any safety rule. The leased permission is automatically revoked as soon as a change in the system status could lead to the violation of the safety rules.

To appreciate the advantage of temporal permissions, consider the *Victim App* in Figure 1(a) that requires the permission P to accomplish its main functionality (e.g., `Location` permission in a navigator app). Permanently revoking of the permission P by the existing approaches makes this app practically useless. However, a careful investigation of the attack scenario makes it clear that the permission P should only be revoked during the "unsafe" time slot. In other words, leasing permission P during the "safe" time slots cannot pose a security risk, yet enables the user to take the full advantage of this app. As a result, an analysis and enforcement approach based on temporal permissions, provides **less disruption** in the normal execution flow of apps.

Another significant advantage of TERMINATOR, attributed to its *permission-based* approach, is the **high coverage** of permission-induced attacks that it can thwart. The existing enforcement techniques only consider certain types of breaches, thereby fail to protect those attacks carried out differently. For instance, according to a recent study [45], the majority of Android security research approaches only consider Intent-based communications to identify inter-component security vulnerabilities, while there are other potentially vulnerable communication methods, such as data-sharing or remote procedure call, which could be exploited by malicious apps. Through meticulous regulation of the common element in all such permission-induced attacks, i.e, permissions, TERMINATOR is able to effectively thwart all of them, regardless of the specific channels exploited by the attackers.

The third distinguishing characteristic of TERMINATOR is its **reliability**. By leveraging the dynamic permissions in Android, our approach avoids any unintended side effects, as it is naturally compatible with the development constraints imposed by the latest versions of Android. Specifically, with the introduction of dynamic permission mechanisms in the latest versions of Android, an app should continue to work properly even if the user does not grant some of the permissions requested by the app [5]. The app in such a case, of course, performs in a downgraded mode, i.e., with some functionalities disabled. Here, we leverage the same feature to revoke an unsafe permission without risking app failure.

4 TERMINATOR

In the previous section, we introduced the idea of using temporal permissions to provide an effective, yet non-disruptive, defense against permission-induced attacks. This section describes how we realized this idea using Android's dynamic permission mechanism.

4.1 Approach Overview

Figure 4 depicts a high-level overview of TERMINATOR, comprised of two phases: *Analysis*, and *Enforcement*. The analysis phase runs once for a set of apps and identifies the potential security risks threatening the Android System (risk detection). The enforcement phase runs continuously and prevents the security threats to occur at run-time (risk prevention). The enforcement components are deployed as an Android app embedded in the device, while the analysis components are deployed externally.

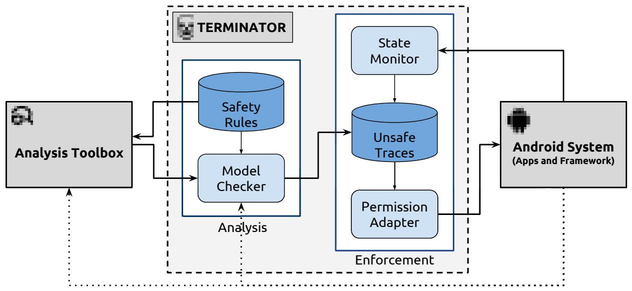


Figure 4: Overview of TERMINATOR framework

To identify the potential security threats, TERMINATOR relies on the state-of-the-art static analysis tools [37], represented as *Analysis Toolbox* in Figure 4. Recall from Section 3.3 that the safety rule formulas include the specification of security vulnerabilities. In TERMINATOR, Analysis Toolbox is responsible for analysis of the installed apps and detecting any instance of the atomic propositions (listed in Figure 2) constituting the security vulnerabilities of the given $SafetyRules$.

As discussed in Section 1, it is overly conservative to assume that identified security risks could be realized in all states of the Android system. To accurately identify the exact conditions under which the identified risks can be realized, TERMINATOR relies on a temporal *Model Checker*, and tracks down any counterexample violating the safety rules with respect to the state transitions of the system. The analysis results are then stored in the *Unsafe Traces* database.

Two components are involved in the enforcement of safety rules: *State Monitor* and *Permission Adapter*. State Monitor keeps track of the system states, particularly those affecting the security properties of the system, and attempts to match the current state of the system against the Unsafe Traces provided by Model Checker. Upon detecting the possibility of a security attack, *Permission Adapter* adopts appropriate countermeasures to prevent the attack from occurring. For this purpose, Permission Adapter refrains to lease the requested permissions enabling the attack, or revokes the previously leased permissions. Once the system moves to a safe state, the adapter re-grants the previously revoked permissions.

In the next two sections, we provide a more detailed description of the components involved in the analysis and enforcement phases.

4.2 Analysis

The set of security risks identified by the Analysis Toolbox are only realizable in specific system states. The analysis phase of TERMINATOR involves identifying those state transitions of the Android system that lead to realization of the security risks.

Recall from Section 3 that we modeled an Android system as a transition system $M = (S, I, R, L)$ and formulated the safety conditions of the system as a set of temporal rules, i.e., $SafetyRules$. Given an Android system M, and a temporal safety rule $r \in SafetyRules$, TERMINATOR is intended to ensure $M \models r$. For this purpose, the *Model Checker* attempts to find any violations, in terms of counterexamples, of the temporal rules.

To realize the Model Checker component, TERMINATOR uses TLC [54], which is intended to check the specifications written in TLA+. TLA, or Temporal Logic of Actions, and its extension TLA+, are originally designed to provide a simple and practical language for high-level specifications of concurrent and distributed systems [34]. TLA specifies the behavior of a system as a sequence of states, where each state is an assignment of values to variables. To model the state transitions, TLA defines next-state relation describing how variables are changed in each step. For this purpose, it uses the primed variable to represent the value of the variable in the next state.

For instance, consider the following TLA+ formula, defined in TERMINATOR to model the state transition of the Android system that occurs due to granting the permission *perm* to an *app*:

$$\begin{aligned}&Grant(app, perm) \triangleq\\ &\wedge perm \in RequestedPerms[app]\\ &\wedge permStat[\langle app, perm\rangle] = \text{“}Revoked\text{”}\\ &\wedge permStat' = [permStat\ EXCEPT\ ![\langle app, perm\rangle] = \text{“}Granted\text{”}\\ &\wedge \text{UNCHANGED}\ appStat\end{aligned} \tag{5}$$

In this TLA+ formula, two variables are used to model the state of the system: *permStat* representing the current state of the permission configuration for each app, and *appStat* specifying the state of each individual app. Types of those two variables are formally defined via the following type invariant assertion:

$$\begin{aligned}&TypeInvariant \triangleq\\ &\wedge appStat \in [Apps \rightarrow \{\text{“}Running(Active)\text{”}, \text{“}Terminated\text{”}\}]\\ &\wedge permStat \in [(Apps \times Perms) \rightarrow \{\text{“}Granted\text{”}, \text{“}Revoked\text{”}\}\end{aligned} \tag{6}$$

Formula 5 defines the *Grant* action as an operator with two arguments, *app* and *perm*. The first two lines of the formula specify the state conditions *enabling* the Grant action: (1) *perm* should be among the requested permissions of the *app*, and (2) *perm* should not be previously granted to the *app*. If both conditions are satisfied in a state, the system can move from that state to the next state described in the third line of the formula.

Using the action formulas, such as formula 5, the *Next* state of the system is defined as the disjunction of all possible next-state actions:

$$\begin{aligned}&Next(app, perm) \triangleq\\ &\exists\, app \in Apps : Run(app) \vee Terminate(app) \vee\\ &\exists\, perm \in Perms : Grant(app, perm) \vee Revoke(app, perm) \vee \ldots\end{aligned} \tag{7}$$

Note that in this formula only four next-state actions, including *Grant* action defined in formula 5, are shown. To see the full list of TLA+ next-state actions defined for TERMINATOR, refer to our online documentation [4].

Finally, the specification of Android system is formulated as follows:

$$Spec \triangleq Init \wedge \Box[Next]_{\langle appStat, permStat\rangle} \tag{8}$$

In this TLA+ formula, *Init* represents the initial state of the system (not shown here), where all apps are inactive and all permissions are revoked.

According to the specification formula, there are two possibilities for the next state of a system state: (1) either one of the actions incorporated in the *Next* formula (Formula 7) will take place, or (2) the state variables, namely *appStat* and *permStat*, will leave unchanged.[6]

In addition to modeling the behavior of an Android system using the above formulas, Model Checker should reason about the security properties of the system. More specifically, Model Checker should identify any sequence of actions (i.e., state transitions) that leads to the violation of safety rules. For this purpose, the following theorem is defined as the system invariant needed to be checked:

$$\text{THEOREM}\ Spec \Rightarrow TypeInvariant \wedge \bigwedge_{rule \in SafetyRules} \tag{9}$$

According to theorem 9, given the specification of an Android system (*Spec*), all system behaviors should satisfy the safety rules, such as Rules 1–4 defined in Section 3.

[6] In TLA+, those steps leaving state variables unchanged are called *stuttering* steps

In this formula, *TypeInvariant* is also added to the theorem to ensure that the model checker only explores the valid states of the system, as specified by definition 6.

Having the specification of Android system and safety rules in TLA+, TLC model checkers verifies formula 9.

For this purpose, TLC explores reachable states, looking for any unsatisfying safety rules. In case of finding a violation, it reports the minimal-length trace from an initial state that leads to an unsafe state. The unsafe traces are then stored in the *Unsafe Traces* database in the Android device.

4.3 Enforcement

State Monitor keeps track of the Android system states, looking for violating traces that match any of the traces stored in the database. The monitor component is realized as an Android app that uses Xposed module for collection of runtime data [6]. The Xposed module instruments the root process of Android, without making any changes in the apps' APK files. The implemented module intercepts those events corresponding to TLA+ action operators defined in Section 4.2. Examples of monitored events include but not limited to: granting/revoking of a permission via a permission request dialog, granting/revoking of a permission via system settings, and running or terminating an app.

If the State Monitor finds a match, it marks the matching unsafe trace in the database, which triggers the *Permission Adapter* component. The goal of this component is to regulate the permission configuration of apps such that the system remains in a safe state. Since the identified security risks are all permission-induced, it is sufficient to revoke the corresponding risk-enabling permissions to thwart the attack.

In its effort to thwart an attack, *Permission Adapter* may encounter a situation in which there are multiple candidate permissions for revocation. For instance, consider the inter-app data leak, where a sensitive data protected by the *source* permission in App1 is leaked through a sensitive channel protected by the *sink* permission in App2. In this case, permission Adapter has two choices, since revoking either of the source or sink permissions would prevent the leak form happening. To provide an *effective* yet *non-disruptive* defense against permission-induced attacks, Permission Adapter applies the following method to select the best permission.

It first calculates two scores for each candidate permission:
Risk score that reflects the number of attacks enabled by granting the permission. A permission with high involvement in the identified security threats would have a higher Risk Score. The risk score is calculated based on the analysis results of Model Checker. **Usage score** that indicates the usage frequency of the app requesting the permission. If the permission is requested by an app that is highly used by the user, that permission would receive a high usage score. Unlike the risk score, the usage score is based on user behavior and calculated by the State Monitor component using the Android's `USAGE_STATS_SERVICE` APIs.

Afterwards, Permission Adapter selects the permission with the highest *Revoke Score*, which is calculated as a function parameterized by both the Risk Score and the reverse of the Usage Score, i.e., $\mathcal{F}(RiskScore, UsageScore^{-1})$. In other words, to prevent a security risk, Permission Adapter revokes permissions with higher security risks that are requested by less-frequently-used apps.

Since TERMINATOR is not aware of the user's context, in certain situations the user may disagree with the way in which it prioritizes permissions for revocation. This might happen, for example, when the user anticipates using a rarely used app. Our implementation allows the user to override the TERMINATOR's decision by adding exception rules. Such rules exclude specific app permissions from being revoked, even if they violate the safety rules.

5 EVALUATION

Our evaluation of TERMINATOR addresses the following research questions:

RQ1. *Coverage*: How does TERMINATOR compare against alternative approaches in preventing the variety of permission-induced attacks?
RQ2. *Disruption*: How effective is TERMINATOR in reducing the unnecessary disruptions due to unavailability of permission-protected app functionality?
RQ3. *Applicability* & *Reliability*: What percentage of Android apps are compatible with TERMINATOR? Does the temporal enforcement of TERMINATOR cause any unexpected behaviors?
RQ4. *Performance*: What are the performance characteristics for each phase of TERMINATOR?

5.1 RQ1: Coverage

For a thorough evaluation, we compared the coverage of TERMINATOR with the other state-of-the-art approaches, enumerated in Table 1 under the "*Alternative Approaches*" column. We considered two criteria in selecting other approaches for our comparative analysis. First, the approach should support both detection and prevention of security attacks.[7] Second, the approach should provide a publicly available tool suite. In accordance with the above criteria, we selected three alternative approaches intended to prevent permission-induced security attacks, namely SEPAR [12], SEALANT [35], and DELDroid [29]. SEPAR enforces fine-grained security policies, synthesized by a SAT-based constraint solver, to prevent capability leaks. SEALANT extends Android framework to provide an interceptor that blocks potentially malicious intents. Finally, DELDroid uses a multiple-domain matrix to eliminate the security vulnerabilities violating the least-privilege property of the system.

To eliminate bias in favor of TERMINATOR, we built a collection of subject apps consisting of the apps used in the evaluation of the three mentioned prior approaches as well as a reputable benchmark collection, namely DroidBench [8]. The resulting dataset consisted of a collection of 255 subject apps with known security issues. Out of this collection of apps, we selected those that target Android 6.0 (API level 23) or newer versions. Older versions of the Android framework provide just a static permission model and do not allow users to dynamically grant or revoke permissions at run-time. We then ended up with a total of 69 apps suitable for our experiments.

To evaluate the extent TERMINATOR can prevent security attacks, we executed the attack scenarios from our dataset on an Android phone running TERMINATOR. Recall from Section 4, TERMINATOR relies on static analysis tools to identify the potential security threats. In our experiments, we used a combination of two static analysis tools, namely FlowDroid [8] and IC3 [41], that have also been used in the construction of three prior approaches to which we compare.

Table 1 shows the result of assessing the effectiveness of TERMINATOR compared to the state-of-the-art techniques. The first three columns of Table 1 show the attack scenarios, their source dataset, and the permissions involved in the attack scenarios, respectively.

[7] Approaches such as IccTA [36], JITANA [51], COVERT [13] DialDroid [15], etc. are excluded in our study as they only perform detection, not prevention.

Table 1: Ability of TERMINATOR in preventing permission-induced attacks in comparison with alternative approaches.

#	Permission-Induced Attack Type (Subtype)	Data Set	Involved Permissions	TERMINATOR	Alternative Approaches		
					SEPAR[12]	DELDroid[29]	SEALANT[35]
1	Custom Permission	DD	STORAGE, SMS	⊠	□	□	□
2	Privilege Escalation	DB	LOCATION, STORAGE	⊠	⊠	⊠	⊠
3	Passive Content Leak (CP)	DD	STORAGE, SMS	⊠	□	□	□
4	Passive Content Leak (CP)	DD	SMS	⊠	□	□	□
5	Passive Content Leak (CP)	DD	STORAGE, SMS	⊠	□	□	□
6	Privilege Escalation	DB	READ_PHONE_STATE, STORAGE	⊠	⊠	⊠	⊠
7	Custom Permission	DD	STORAGE, SMS	⊠	□	□	□
8	Privilege Escalation (AH)	DD	STORAGE, SMS	⊠	□	⊠	⊡
9	Privilege Escalation (DCL)	DD	STORAGE, SMS	⊠	□	⊠	⊡
10	Custom Permission	DD	SMS	⊠	□	□	□
11	Passive Content Leak (CP)	DD	STORAGE, SMS	⊠	□	□	□
12	Privilege Escalation (PR)	SP	WAKE_LOCK	⊠	□	⊠	□
13	Privilege Escalation (PR)	DD	SET_WALLPAPER	⊠	□	⊠	□
14	Custom Permission	DD	STORAGE, SMS	⊠	□	□	□
15	Privilege Escalation	SL	SMS	⊠	⊠	⊠	⊠
16	Passive Content Leak (CP)	DD	STORAGE, SMS	⊠	□	□	□
17	Privilege Escalation (BT)	SP	WAKE_LOCK	⊠	□	⊠	□
18	Privilege Escalation	SL	LOCATION	⊠	□	⊠	⊠
19	Privilege Escalation (SH)	DD	SMS	⊠	□	⊠	⊠
20	Privilege Escalation (PR)	DD	LOCATION, SMS	⊠	⊠	⊠	⊡
21	Privilege Escalation	SL	CONTACTS	⊠	□	⊠	⊠
22	Custom Permission	DD	STORAGE, SMS	⊠	□	□	⊠
23	Privilege Escalation (PR)	DD	LOCATION, SMS	⊠	⊠	⊠	⊠
24	Privilege Escalation (MAL)	DD	SMS	⊠	⊠	⊠	□
25	Privilege Escalation	DB	LOCATION, STORAGE	⊠	⊠	⊠	⊠
26	Privilege Escalation	SL	LOCATION	⊠	□	⊠	□
27	Privilege Escalation (PR)	DD	SMS	⊠	□	⊠	⊠
28	Privilege Escalation	DB	READ_PHONE_STATE, STORAGE	⊠	⊠	⊠	⊠
29	Passive Content Leak (CP)	DD	SMS	⊠	□	□	□
30	Privilege Escalation	DB	READ_PHONE_STATE, STORAGE	⊠	⊠	⊠	⊠
31	Privilege Escalation (PR)	DD	SMS	⊠	⊠	⊠	□
32	Privilege Escalation (PR)	DD	LOCATION	⊠	□	⊠	⊠
33	Privilege Escalation (DCL)	DD	STORAGE, LOCATION	⊠	□	⊠	□
34	Passive Content Leak (CP)	DD	STORAGE, SMS	⊠	□	□	□
35	Privilege Escalation (AH)	DD	STORAGE	⊠	□	⊠	□
36	Custom Permission	DD	STORAGE, SMS	⊠	□	□	□
37	Privilege Escalation	DB	READ_PHONE_STATE, SMS	⊠	⊠	⊠	⊠
38	Privilege Escalation (AH)	DD	STORAGE	⊠	□	⊠	□
39	Privilege Escalation	DB	ACCESS_FINE_LOCATION	⊠	⊠	⊠	⊠
40	Privilege Escalation	DB	LOCATION, STORAGE	⊠	⊠	⊠	⊠
41	Custom Permission	DD	SMS	⊠	□	□	□
Total thwarted attacks				41	13	27	16
Coverage (true-positive rate)				100%	31.7%	65.9%	39.0%

⊠(□): attack scenario is (not) prevented by the approach, ⊡: the approach crashed during the analysis
Attack Subtypes: PR: Permission Re-Delegation, AH: Activity Hijack, SH: Service Hijack, BT: Broadcast Theft, MAL: Malicious Activity Lunch, DCL: Dynamic Class Loading, Content Provider (CP)
Data-sets: DD: DELDroid [29], SL:SEALANT[35], SP:SEPAR[12], DB: DroidBench [8]

The other columns indicate whether each of the four approaches assessed was successful in preventing the attack (⊠) or not (□, ⊡).

According to the results, TERMINATOR is able to prevent all the attack scenarios with no false negatives. The success rate of the other techniques in preventing the permission-induced attacks ranges from 31.7% to 65.9%. A detailed look at Table 1 indicates that most of the missing attacks are those whose detection requires temporal analysis. For instance, consider the attack scenario #36, where a malicious application has defined a custom permission identical to the permission defined by a vulnerable app to protect its internal database. As a result, the malware can illegally access sensitive information stored in the vulnerable app. This vulnerability, however, is only exploitable if the malware is installed *before* the victim app. Thereby, all prior non-temporal approaches fail to detect such attacks. To tackle this issue, a conservative approach might prevent the aforementioned attack by permanently revoking database access of the victim app. This approach, however, would cause unnecessary disruptions, particularity when the vulnerability is not exploitable, i.e., the victim app is installed before malware in this case. In the next research question (RQ2), we investigate the consequences of permanently revoking the permissions of vulnerable apps through additional experiments.

5.2 RQ2: Disruption

For this research question, we focus on alternative permission-based enforcement techniques. Generally speaking, permission-based security enforcement can be applied at install-time or run-time [26]. An install-time approach prevents the installation of vulnerable apps, while a run-time approach revokes the permissions upon identification of an attack scenario. Run-time approaches can further be either permanent, whereby the permission decisions are final, or temporal, as in the case of TERMINATOR, whereby the permission decisions are adjusted over time. Since the prior tools implementing the competing techniques are either not available, as is the case with AppFence [31] and AppGuard [9], or outdated and inapplicable, as is the case with Kirin [25], we implemented both install-time and permanent-run-time enforcement approaches described in the prior work to compare against TERMINATOR's enforcement strategy.

To evaluate the level of disruption due to unavailability of permission-protected app functionality, we needed access to legitimate use-cases for apps in our dataset. Attack scenarios used in the evaluation of RQ1 are not representative of the apps' functional use-cases; thereby, they are not suitable for evaluating the level of disruption caused by the revocation of app permissions. To that end, we followed a semi-systematic approach to extract functional use-cases for the vulnerable apps in our dataset. We first downloaded the description of subject apps from the app markets (Google Play or F-Droid). We then asked a group of graduate students to construct, if possible, functional use-cases for each sentence or bullet in the app description. Additionally, we used available system tests for open-source subject apps as another source for identifying the legitimate use-cases. In total, we were able to derive 186 legitimate use-cases for subject apps in this research question. The full set of use-cases and subject apps are publicly available on the project website [4].

To measure the disruptions caused by the two run-time approaches, we first executed the attack scenarios from Table 1 to instigate an enforcement decision, i.e., force the approach to adjust the permission configuration. Subsequently, we ran the legitimate use-cases involving the apps in the attack to determine if the use-cases can be executed successfully or not. Table 2 summarizes the results of comparing different enforcement strategies for permission-based approaches. The first column shows the subject apps. The

Table 2: Efficacy of permission-based techniques in reducing the unnecessary disruptions.

Vulnerable Apps	#Use-cases	#Allowed Scenarios		
		Run-time		Install-time
		Temporal	Permanent	
de.*.geobookmark	3	3	0	0
com.*.multismssender	8	8	4	0
com.*.calendar	14	14	6	0
com.*.smsscheduler ↯	9	9	5	0
org.*.trackbook	9	9	2	0
com.*.simpledeadline	13	13	9	0
com.getback_gps	10	10	6	0
com.*.camera	12	12	0	0
com.*.gallery	4	4	3	0
com.*.manager	11	11	5	0
com.*.anki	18	18	0	0
com.*.screennotification	3	3	2	0
com.*.notes	2	2	1	0
cz.*.forcastie	7	7	4	0
com.*.loginexample	2	2	1	0
com.*.sms	3	3	1	0
com.*.opps_wrong_tab	8	8	7	0
code.*.sendsmstest	3	3	1	0
org.*.myexpenses	24	24	8	0
com.*.ukweather	4	4	0	0
com.*.client	8	8	7	0
fr.*.ommons	11	11	9	0
Total allowance		186	81	0
Disruption (false-positive rate)		0%	56.45%	100%

second column shows the number of legitimate use-cases for the subject apps. The last three columns show the number of use-cases allowed by each approach. The results from this analysis confirm that the run-time-temporal approach adopted in TERMINATOR outperforms other enforcement techniques in terms of unnecessary disruption, i.e., false-positive rate. The install-time enforcement approach performs worst (100% false positive), as it does not allow the installation of a vulnerable app. The run-time-permanent approach (with 56% false positive) on the other hand, allows installation, yet revokes unsafe permissions permanently. Therefore, some of the legitimate permission-protected use-cases can never execute after revocation, even in the absence of a security threat.

For example, the security analysis performed by TERMINATOR identified *GetbackGPS* app (com.getback_gps in Table 2) as being vulnerable to privilege escalation attack—attack scenario #20 in Table 1, whereby its sensitive location information can be leaked. This vulnerability is only exploitable if two conditions are satisfied simultaneously: (1) a malware app with access to a sink channel (e.g., SMS) is installed and *running* on the phone, and (2) the malware has been *granted* the sink permission. Since the app is vulnerable, the install-time approach simply does not allow its installation to avoid any chance of leaking user's location information. The run-time-permanent approach on the other hand, allows the installation of GetbackGPS, yet permanently revokes its `LOCATION` permission to remove the vulnerability. Our run-time-temporal enforcement approach, however, leases `Location` permission to GetbackGPS, as long as the above conditions are not satisfied, during which all of the legitimate use-cases of the app are available.

5.3 RQ3: Applicability & Reliability

5.3.1 Applicability. Recall from Section 4.3, TERMINATOR relies on the dynamic permission mechanism, supported by Android 6 and newer versions of the framework, to regulate app permissions at run-time. However, not all the apps available on the Android marketplace are compatible with the new versions of Android. To investigate the extent to which TERMINATOR is applicable to Android apps, we measured percentage of the apps on the official Android marketplace, i.e., Google Play, that target API level 23 (Android 6) and above.

To that end, we randomly collected 48,795 apps from different app categories, and distinguished Android-6-compatible apps by examining the `targetSdkVersion` tag specified in their *manifest* file. To avoid any bias in the results, we did not use any particular criteria, such as high popularity or high ranking, in selection of the apps to be analyzed. Table 3 demonstrates percentage of the apps targeting API level 23 and above among the apps collected from 15 different app categories of the Google Play repository.According to the results, on average 89.8% of the G.Play apps support dynamic permissions.

To further investigate the support for dynamic permissions among popular apps, we also collected top 100 popular apps on Google Play. As shown in the last column of Table 3, all of the top 100 apps on Google Play support dynamic permissions, thereby are compatible with TERMINATOR. These results indicate that a large majority of the apps on the Android official marketplace can benefit from TERMINATOR for run-time security enforcement.

5.3.2 Reliability. Although the majority of collected apps support Android 6 and above, it is possible that they do not properly handle dynamic permissions. Failing to adjust the functionality of an app to dynamic permissions can lead to unexpected behaviors, e.g., app crashes if the user decides to revoke a permission. Hence, we also need to investigate the reliability of adopting an approach like TERMINATOR, which revokes permissions at run-time.

To investigate reliability of TERMINATOR, we recorded Logcat outputs during the execution of both the attack scenarios and canonical use-cases for subject apps discussed in RQ1 and RQ2. We later explored collected logs, searching for any crash messages due to improper handling of dynamic permissions. Out of the 69 subject apps in our dataset, we found one app, *SMS Scheduler* (marked with ↯ in Table 2), that crashes due to the permission revocation.

From this data—low percentage (around 1.5%) of apps crashing when revoking their permissions and high percentage (around 89.8%) of app compatibility with recent versions of Android—we conclude that TERMINATOR can reliably be applied to a large majority of Android apps available on the market.

5.4 RQ4: Performance

To examine the performance characteristics of TERMINATOR, we measured the execution time taken for each phase of TERMINATOR, i.e., *analysis* and *enforcement* . We performed our experiments on a PC with an Intel Core i7 2.4 GHz CPU processor and 16 GB of main memory for the analysis phase, and a Nexus 5x phone operated by the Android framework version 6 for the enforcement phase.

TLC is configurable in two operating modes, *simulation* and *model-checking*. In the simulation mode, TLC verifies the system behavior up to a fixed number of system states. In the model-checking mode, on the other hand, there is no limit for the number of states to be explored. Applying an upper bound over the state exploration may lead to the possibility of missing attacks concealed within states not explored. We configured TLC to operate in the simulation mode to guarantee the termination of the analysis phase. This guarantee is required for TERMINATOR given that the reachable states of our model for the Android system is infinite. In our experiments, TERMINATOR was able to identify all of the attack scenarios (see Table 1). For these attack scenarios, TLC took at most 7 seconds to find the attack through the exploration of over 707,000 states.

To determine the performance of the enforcement phase, we calculated the overhead of running Monitor and Adapter components of TERMINATOR during the execution of 227 (41 attack scenarios and 186 canonical use-cases) scenarios exercised in RQ1 and RQ2. We repeated the execution of each scenario 5 times to ensure 95% confidence interval for the reported values. According to our experiments, the run-time overhead of TERMINATOR enforcement phase

Table 3: Percentage of Android-6-compatible apps in Google Play

Randomly Selected by Category															Average	Top 100
Art	Books	Finance	Food	Health	Maps	Music	News	Photo	Shopping	Social	Tools	Video	Weather	Game		
100%	78.9%	86.6%	96.1%	85.9%	92.2%	89.1%	90.4%	89.9%	94.6%	90.9%	86.5%	86.1%	88.2%	91.6%	89.8%	100%

is 714±33 milliseconds on average for each use-case. Given that the average execution time for each use-case is 12 seconds, this overhead is negligble, as it is less than the threshold users can perceive slowness in an app, according to offical Android documentation [2]. Note that the analysis phase is performed once per system configuration, while the enforcement component runs continuously as the user interacts with the apps.

6 RELATED WORK

We provide a discussion of the related efforts in light of our work.

Assessment of Security Properties. A large body of research focuses on performing security analysis in the context of Android. Using a taxonomy, Sadeghi et al. [45] classified over 300 research papers for security assessment of Android apps. According to this taxonomy, our work is classified under the group of research [7, 10, 13, 18, 23, 27, 49] that leverages *formal analysis* to reason about security properties of a system.

Several approaches [18, 27, 49] under this category leverage formal methods to abstract the semantics of application code and express the security properties of the apps. Using the derived model of the program, this group of techniques are able to identify security issues such as data leakage. Another group of formal approaches [7, 10, 11, 13, 23] perform the security analysis at a higher level of abstraction— i.e., architectural level. This thrust of research model an Android system as a set of components that communicate with each other through connectors. Working at this level of abstraction allows the research to detect more involved security attacks, such as privilege escalation attacks or app collusion. In contrast to TERMINATOR, all of the prior formal approaches fail to incorporate the notion of time in their model of Android permissions— that is, they assume all requested permissions are permanently granted to an app as soon as the user accepts to install the app.

Enforcement of Security Policies. In addition to the security analysis, TERMINATOR enforces policies at run-time. Similar to our work, a plethora of approaches [12, 24, 33, 35, 44, 47, 52, 53] have been constructed to enforce security policies in Android devices. In their survey of security solutions for Android, Sufatrio et al. [50] categorized this group of research based on the security attacks they can prevent. In another study, Heuser et al. [30] classified this group of research based on the authorization hook semantics.

To enhance the security of the system, the proposed techniques place additional constraints on different elements of Android system, from inter-component communication (ICC) [12, 35] to network access [33, 39, 55], and Content Providers [14, 57]. Hence, each technique might fail to protect those attacks carried out through the other channels that are not hooked by the approach. For instance intent-based preventive methods [12, 35] are not able to prevent inter-app security attacks exploiting unprotected content providers. By identifying and thwarting the root cause of all permission-induced attacks, i.e. unsafe permission, TERMINATOR is able to prevent all of them, regardless of the type of vulnerable elements being exploited.

Moreover, due to modification of Android framework [17, 29, 31, 43, 52] or individual apps [9, 19, 44, 53], the prior approaches are prone to place the app in an unexpected state or even cause it to crash. TERMINATOR avoids any such side effect as it relies on dynamic permission mechanism, which is officially provided by the Android framework, and adopted by app developers.

Enhancement of Permission Model. The other relevant thrust of research has focused on enhancing the permission model of Android, in order to mitigate the risk associate with permission misuse. A comprehensive study of permission-induced security issues, along with the proposed countermeasures are provided by Fang et al. [26]. Coarse granularity is a shortcoming of Android permissions discussed in several research papers. To overcome this issue, several finer-grained implementation of permissions have been proposed [21, 25, 32, 42, 53]. Dissimilar to this group of the approaches that limit the scope of permissions to specific components, TERMINATOR restricts the availability of permissions over the time dimension.

To enhance the permission mechanism of Android, several approaches [14, 31, 38, 40, 57] provide easy-to-use interfaces that allow users to selectively grant permissions to apps. Beside the fact that such feature is officially supported by the Android version 6 and above, enabling users to revoke permissions, either via Android's build-in interface or through the academic approaches, is not a solution for security enhancement of the system. Rather, it shifts the problem from installation-time to run-time. The users still do not know when it is safe to grant a requested permission to an app. Using temporal permission, TERMINATOR lets user take full advantage of app functionality while protecting them against permission-induced attacks.

7 CONCLUSION AND FUTURE WORK

We presented a permission analysis and enforcement framework that, in contrast to the prior work, considers the temporal aspects of permission-induced attacks for their detection and prevention. The framework, called TERMINATOR, is realized in two phases. In the analysis phase, it uses a temporal logic model checker to identify the security risks with respect to dynamic states of the system. In the enforcement phase, it relies on Android's dynamic permission mechanism to prevent the identified security threats from materializing by regulating the permission configuration of the system. Our evaluation results indicate that TERMINATOR is able to provide an effective, yet non-disruptive, defense against permission-induced attacks. The results also show that our approach, which is implemented without modification of Android framework or implementation logic of apps, is highly reliable and compatible with the great majority of Android apps available on the marketplace.

In this work, temporal rules are enforced at the app level, meaning that permissions are leased to the whole app. A more fine-grained temporal rule, that reduces an app's attack surface further through leasing permissions to a subset of its components, is potentially an interesting avenue of future research.

ACKNOWLEDGEMENT

This work was supported in part by awards CCF-1252644, CNS-1629771 and CCF-1618132 from the National Science Foundation, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

REFERENCES

[1] 2014. NVD:CVE-2014-8609. (2014). https://nvd.nist.gov/vuln/detail/CVE-2014-8609

[2] 2017. Keeping your app responsive. (2017). https://developer.android.com/training/articles/perf-anr.html

[3] 2017. PendingIntent. (2017). https://developer.android.com/reference/android/app/PendingIntent.html

[4] 2017. Terminator web page [In accordance with the double-blind policy]. (2017). https://sites.google.com/view/terminator18

[5] 2017. Working with System Permissions. (2017). https://developer.android.com/training/permissions

[6] 2017. Xposed Framework. (2017). http://repo.xposed.info/

[7] Alessandro Armando, Gabriele Costa, and Alessio Merlo. 2012. Formal Modeling and Reasoning about the Android Security Framework. In *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*. 64–81.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 259–269.

[9] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard - Enforcing User Requirements on Android Apps. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 543–548.

[10] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2015. Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. 73–89.

[11] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2017. A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing* (2017). https://doi.org/10.1007/s00165-017-0445-z

[12] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand Behrouz, and Sam Malek. 2016. Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. 514–525.

[13] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Trans. Software Eng.* 41, 9 (2015), 866–886.

[14] Alastair R. Beresford, Andrew C. Rice, Nicholas Skehin, and Ripduman Sohan. 2011. MockDroid: trading privacy for application functionality on smartphones. In *12th Workshop on Mobile Computing Systems and Applications, HotMobile'11, Phoenix, AZ, USA, March 1-3, 2011*. 49–54.

[15] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*. 71–85.

[16] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.

[17] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. 2011. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische UniversitÃd't Darmstadt, Technical Report TR-2011-04* (2011).

[18] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. 47–62.

[19] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R. Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. 2013. Contextual Policy Enforcement in Android Applications with Permission Event Graphs.. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*.

[20] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David A. Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*. 239–252.

[21] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. 2010. CRePE: Context-Related Policy Enforcement for Android. In *Information Security - 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers*. 331–345.

[22] Maxwell John Cresswell and George Edward Hughes. 2012. *A new introduction to modal logic*. Routledge.

[23] Mads Dam, Gurvan Le Guernic, and Andreas Lundblad. 2012. TreeDroid: a tree automaton based approach to enforcing data processing policies. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 894–905.

[24] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*.

[25] William Enck, Machigar Ongtang, and Patrick D. McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. 235–245.

[26] Zheran Fang, Weili Han, and Yingjiu Li. 2014. Permission based Android security: Issues and countermeasures. *Computers & Security* 43 (2014), 205–218.

[27] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 576–587.

[28] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. 2012. Modeling and enhancing androidâĂŹs permission system. In *European Symposium on Research in Computer Security*. Springer, 1–18.

[29] Mahmoud Hammad, Hamid Bagheri, and Sam Malek. 2017. Determination and Enforcement of Least-Privilege Architecture in Android. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*. 59–68.

[30] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 1005–1019.

[31] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. These Aren'T the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. 639–652.

[32] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*. 3–14.

[33] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*. 775–792.

[34] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc.

[35] Youn Kyu Lee, Jae Young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A *SEALANT* for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 312–323.

[36] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 280–291.

[37] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information & Software Technology* 88 (2017), 67–95.

[38] Kurt Mueller and Kevin Butler. 2011. Poster: Flex-p: flexible android permissions. In *Proc. of IEEE S&P*.

[39] Adwait Nadkarni and William Enck. 2013. Preventing accidental data disclosure in modern operating systems. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 1029–1042.

[40] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*. 328–332.

[41] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick D. McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 77–88.

[42] Machigar Ongtang, Stephen E. McLaughlin, William Enck, and Patrick D. McDaniel. 2012. Semantically rich application-centric security in Android. *Security and Communication Networks* 5, 6 (2012), 658–673.

[43] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David A. Wagner. 2012. AdDroid: privilege separation for applications and advertisers in Android. In *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*. 71–72.

[44] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*. 40–49.

[45] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2017. A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software. *IEEE Trans. Software Eng.* 43, 6 (2017), 492–530.

[46] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. 2011. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

[47] Daniel Schreckling, Joachim Posegga, Johannes KÃűstler, and Matthias Schaff. 2012. Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android. In *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems - 6th IFIP WG 11.2 International Workshop, WISTP 2012, Egham, UK, June 20-22, 2012. Proceedings*. 208–223.

[48] Wook Shin, Sanghoon Kwak, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. 2010. A Small But Non-negligible Flaw in the Android Permission Scheme. In *POLICY 2010, IEEE International Symposium on Policies for Distributed Systems and Networks, Fairfax, VA, USA, 21-23 July 2010*. 107–110.

[49] Fu Song and Tayssir Touili. 2014. Model-Checking for Android Malware Detection. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. 216–235.

[50] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. 2015. Securing Android: A Survey, Taxonomy, and Challenges. *ACM Comput. Surv.* 47, 4 (2015), 58:1–58:45.

[51] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-an, Gregg Rothermel, and Jackson Dinh. 2017. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 324–334.

[52] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society.

[53] Rubin Xu, Hassen SaÃŕdi, and Ross Anderson. 2012. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 27–27.

[54] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA$^+$ Specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. 54–66.

[55] Zhibo Zhao and Fernando C. Colón Osorio. 2012. TrustDroid: Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*. 135–143.

[56] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*.

[57] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. 2011. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*. 93–107.