Self-Protection of Android Systems from Inter-component Communication Attacks

Mahmoud Hammad, Joshua Garcia, and Sam Malek
Department of Informatics
University of California, Irvine
Irvine, CA, USA
{hammadm, joshug4, malek}@uci.edu

ABSTRACT

The current security mechanisms for Android apps, both static and dynamic analysis approaches, are insufficient for detection and prevention of the increasingly dynamic and sophisticated security attacks. Static analysis approaches suffer from false positives whereas dynamic analysis approaches suffer from false negatives. Moreover, they all lack the ability to efficiently analyze systems with incremental changes—such as adding/removing apps, granting/revoking permissions, and dynamic components' communications. Each time the system changes, the entire analysis needs to be repeated, making the existing approaches inefficient for practical use. To mitigate their shortcomings, we have developed SALMA, a novel self-protecting Android software system that monitors itself and adapts its behavior at runtime to prevent a wide-range of security risks. SALMA maintains a precise architectural model, represented as a *Multiple-Domain-Matrix*, and incrementally and efficiently analyzes an Android system in response to incremental system changes. The maintained architecture is used to reason about the running Android system. Every time the system changes, SALMA determines (1) the impacted part of the system, and (2) the subset of the security analyses that need to be performed, thereby greatly improving the performance of the approach. Our experimental results on hundreds of real-world apps corroborate SALMA's scalability and efficiency as well as its ability to detect and prevent security attacks at runtime with minimal disruption.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*;

KEYWORDS

Self-protecting system;Android security;Software Engineering

ACM Reference Format:
Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. Self-Protection of Android Systems from Inter-component Communication Attacks. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238207

1 INTRODUCTION

Reusability is a major reason behind the meteoric rise in the popularity of the Android platform [12] and the increasing number of apps [10]. To develop rich apps, Android promotes reusability of (1) information and services provided by third-party apps, through its flexible Inter-Component Communication (ICC) model, and (2) sensitive resources protected by a permission-based model. However, since the inception of Android, the ICC and the permission-based models have become the main attack vector for Android apps, which can lead to serious security and privacy risks [23, 32, 42, 57].

The current state-of-the-art security mechanisms for Android apps, both static and dynamic analysis approaches, are insufficient for detecting and preventing the increasingly sophisticated security attacks. Static analysis approaches suffer from false positives due to their over-approximation of the analyzed apps, e.g., producing warnings for vulnerabilities that are not executable at runtime. On the other hand, dynamic analysis approaches suffer from false negatives due to the *reachability* problem, where vulnerabilities are missed due to inputs that fail to reach the vulnerable code.

Moreover, due to the complex and dynamic nature of Android systems (e.g., adding/removing an app, granting/revoking a permission, and dynamic class loading), their security posture changes over time. Simply repeating the entire security analysis of an Android system, either statically or dynamically, every time the system changes is prohibitively expensive for practical use.

To overcome the shortcomings of the current approaches, we have developed SALMA, a novel self-protecting Android software system that (1) continuously monitors the running Android system, (2) incrementally and efficiently analyzes the security posture of the system, and (3) dynamically enforces security policies to prevent security attacks at runtime. SALMA leverages static program analysis to automatically derive the initial abstract representation, i.e., a model, of an Android system. SALMA then monitors the running system to keep the model synchronized with the running system. Whenever the model changes, SALMA determines (1) the impacted part of the system, and (2) the required security analyses that need to be performed. Finally, SALMA adjusts security policies and enforces them at runtime, thus ensuring the system is safe and protected at all times.

SALMA models the system as a Multiple-Domain-Matrix (MDM) [49]—which provides an elegant, yet compact, representation of all relationships among principal elements, such as components and permissions, in a system. Our implementation of the MDM provides a flexible way to load and analyze parts of the system, improving the scalability and efficiency of the overall approach. SALMA can be used to protect Android systems without modification of the apps' implementation logic, allowing our approach to be applied to all existing Android apps. Our evaluation of SALMA using hundreds of real-world apps corroborates its efficiency and scalability in

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238207

analyzing evolving Android systems with minimal disruption to apps and their services while thwarting security threats to keep the system protected at all times. SALMA achieves 70%-84% greater detection of attacks than state-of-the-art approaches and 45%-85% greater prevention of attacks than those approaches.

The rest of this paper is organized as follows. Section 2 describes the security attacks our approach can detect and prevent. The research gap in the current security mechanisms for Android apps is presented in Section 3. Section 4 illustrates an Android system to motivate our research. The approach and its implementation are discussed in Sections 5 and 6, respectively. The evaluation results are presented in Section 7. Finally, the paper concludes with an overview of the related literature and areas of future research.

2 ANDROID SECURITY ATTACKS

Inter-component communication (ICC) in Android is mainly achieved either by sending *Intents* or using Unified Resource Identifiers (URIs). An Intent is a message exchanged among apps, whose payload includes an action to be performed along with the data that supports that action. Component capabilities are then specified as a set of *Intent Filters* that represent the kinds of requests handled by a given component. Component invocations come in different flavors, including explicit or implicit, and intra-app or inter-app. URIs are used to access or manipulate the encapsulated data in `Content Providers`, the database components in Android apps. Android's ICC allows for late run-time binding between components in the same or different apps, where the calls are not explicit in the code, but instead are made through exchanging messages that correspond with certain events, a key property of event-driven systems. Android applies a permission-based model to protect sensitive resources, both system resources and application resources, that each app is allowed to access. Since Android version 6, Google changed the permission management system from static to dynamic, allowing users to grant or revoke permissions at runtime.

The Android ICC interaction mechanism and the current permission model of Android are the root cause of many security vulnerabilities. They have become a vulnerable attack surface of an Android system which threatens user privacy and has affected millions of users [9]. These attacks are widely discussed in the literature [22, 23, 28, 29, 32–34, 42, 44, 64, 70]. ICC attacks are security risks facilitated by (1) incorrectly or maliciously using the message-passing system in Android or (2) misusing the permissions in Android. SALMA provides self-protection against these ICC attacks. This section briefly describes these attacks.

Unauthorized Intent Receipt: In this attack, a malicious component intercepts an implicit Intent by declaring an Intent Filter that matches the sent Intent [23, 44]. In such an attack, a malicious component can access all enclosed data in the intercepted Intent and, possibly perform a phishing attack [13].

Intent Spoofing: In such an attack, a malicious component can communicate with an exported component that is not expecting such communication [23, 44]. If a victim component blindly trusts the received Intent, the malicious component can cause the victim component to perform undesirable actions [36].

Privilege Escalation: This attack allows a malicious component to indirectly perform a privileged task [22, 34]. In this attack, if a vulnerable component possesses a permission without appropriately protecting its interface, a malicious component can perform a privileged task, such as sending a text message or tracking the location of a user, by interacting with that vulnerable component.

Identical Custom Permission: Any Android app can also define its own permissions and use them to protect its components. Each permission must define a name and a *protection level*, where each level affects the extent to which a permission can be granted or revoked. The notable protection levels for this paper are *Normal* and *Signature*. A *Normal* permission is automatically granted to apps that request them without asking for the user's approval. A *Signature* permission is granted to apps that are signed with the same certificate as the app that declared the permission.

The custom permission model of Android contains a vulnerability rooted in its design: "if two apps define the same custom permission, whichever app is installed first is the one whose definition is used" [18]. A malicious app can exploit this vulnerability to access a protected component with a custom permission by declaring another permission with the same name as that legitimate one.

Passive Data Leak: `Content Providers` can be used for both intra-app data persistence as well as sharing data across apps. If the read access to a `Content Provider` is not properly guarded with a permission, other apps can exploit this vulnerability to disclose and leak sensitive data [42].

Content Pollution: This attack is possible when the write access to a `Content Provider` is not properly guarded with a permission [42]. This vulnerability allows a malicious app to manipulate sensitive data managed by a vulnerable app. The manipulated data can cause severe side effects such as altering firewall rules or blocking incoming calls.

3 RESEARCH GAP

The current security mechanisms for Android apps, both static and dynamic analysis approaches, are insufficient for detecting and preventing the increasingly dynamic and sophisticated attacks.

Static analysis approaches [48], [68], [46][50], [19], [70], [23], [54], [34], [31], [47], [20], [60], [37], [38] suffer from false positives, i.e., false alarms. The high number of false alarms generated by such approaches lower their applicability. Moreover, static analysis approaches face severe limitations when it comes to analyzing obfuscated or dynamically loaded code [57], thus in practice also suffer from false negatives. Precise forms of static analysis also require significant amounts of computing resources and can take a substantial amount of time to execute.

Dynamic analysis approaches [30], [66], [17], [40], [27], [55], [21] are not sound, and are thus prone to false negatives. These approaches are susceptible to a variety of anti-debugging and anti-monitoring defenses [15, 24, 26, 35, 43, 56, 58, 59, 62, 67] as well as *time bombs* [25], which further decrease their efficacy. Furthermore, dynamic approaches are tedious and time consuming, as exhaustive execution of apps can take a substantial amount of time.

To overcome the limitation of pure static or pure dynamic analysis, Holla and Katti [39] discussed the need for hybrid Android security approaches. Despite that, few approaches proposed hybrid techniques such as Dr. Android [41], SmartDroid [71], and ProfileDroid [69]. Nevertheless, these tools provide detection capabilities but not prevention mechanisms. Moreover, they require changes to apps' implementation logic which prevent their practical use.

All of these approaches perform complete analysis of Android systems, and hence lack the ability to efficiently analyze systems as changes occur—such as adding/removing apps, granting/revoking permissions at runtime, or dynamically loading code. SALMA mitigates the aforementioned shortcomings through (1) continuously monitoring the running system, (2) incrementally and efficiently analyzing the system against a broad-range of ICC security vulnerabilities, and (3) enforcing adaptation tactics to prevent security attacks at runtime with minimal disruption.

4 ILLUSTRATIVE EXAMPLE

To further motivate our research and illustrate our approach, we provide an example of an evolving Android system that consists initially of two apps: `SuperPhone` and `StayHealthy` apps, illustrated in Figure 1 (a). The `MakeCalls` Activity in the `SuperPhone` allows a user to make phone calls and it stores calls’ information in the `CallsDB`, a Content Provider component. The `History` queries the stored calls in `CallsDB` and lists them to a user. `StayHealthy` is a fitness app that allows users to log their daily workouts, via `Exercises`, and meals, via `Meals`. Both of these Activities are accessible from the `Home` Activity. The `LocTracker` is a service that runs in the background and tracks the user’s location upon receiving an Intent. `Exercises` uses `LocTracker` to draw a route map of a user’s workout. `StayHealthy` also allows a user to share his logged activities, either workouts or meals, with friends by sending text messages. The `Share` Service sends spatial data, i.e., tagged data with the current user’s location, as a text message to the phone number specified in the received Intent. `Share` is being used by both `Exercises` and `Meals` Activities.

The `Share` Service is a vulnerable service since it does not check if the calling component has the appropriate permissions, SMS and Location in this example, before sending spatial data. Whereas the `LocTracker` is a secure Service since it checks for the granted permissions of the caller component. Such a check in Android can be achieved using the `checkCallingPermission` API. Although `Share` is a vulnerable component in the Android system illustrated in Figure 1 (a), the current system is not actively threatened since no component is exploiting this vulnerability.

At a later time, a user installs a new app called `BrainTeaser`, as shown in Figure 1 (b). `BrainTeaser` is a malicious app that challenges a user to solve mathematical questions and then measures her intelligence quotient (IQ). The `IQtest` Activity displays questions and communicates with the `Qgenerator` Service to get the next question. `Qgenerator` is a malicious component that, once started, communicates with the `Share` Service of the `StayHealthy` app. Since `Share` does not check if the caller components has the required permissions, i.e., SMS and LOCATION permissions, this component is vulnerable to a privilege-escalation ICC attack. Therefore, the communication between the `Qgenerator` and `Share` results in exploiting this vulnerability which allows `Qgenerator` to leak the user’s location to any premium rate number without having the proper permissions to perform such a task.

The attack described in this section is a legitimate scenario in the current implementation of the Android platform [19]. Moreover, performing a complete analysis of all Android apps in the system every time the system changes is neither efficient nor practical. We show how, through a runtime monitoring and incremental analysis, SALMA can efficiently and effectively mitigate such a threat.

5 APPROACH

Figure 2 depicts a high-level overview of SALMA—which contains two layers, the *protected layer* and the *protecting layer*. The protected layer consists of our modified Android framework and a set of apps that a user installs on a device. The protecting layer realizes the IBM MAPE-K control loop [45]. MAPE-K consists of four components and a knowledge component. The *Knowledge* contains an architectural model of the system. The *Monitor* observes the system and keeps the model synchronized with the running system. The *Analyzer* assesses the system for security threats. The *Planner* determines the best security policies, a.k.a. adaptation tactics, to be enforced at runtime by the *Executor*.

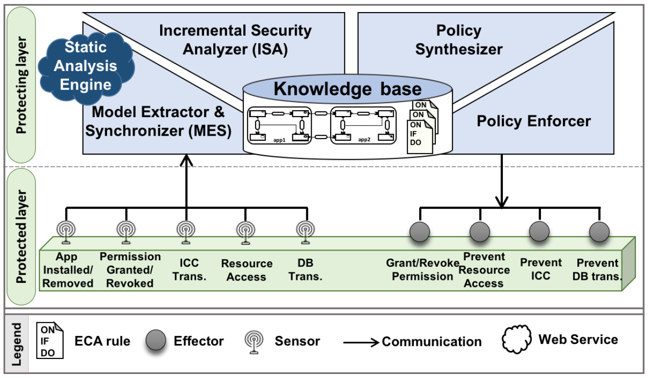


Figure 2: Overview of SALMA.

Figure 2 depicts instantiations of each of the MAPE-K components in the protecting layer: *Monitor Extractor & Synthesizer* (*MES*), which is a *Monitor*; *Incremental Security Analyzer* (*ISA*), which is an *Analyzer*; *Policy Synthesizer*, which is a *Planner*; and *Policy Enforcer*, which is an *Executor*. *MES* automatically obtains and maintains a precise runtime architectural model of an Android system. When a change occurs in the maintained runtime model, the *ISA* (1) determines the impacted part of the system due to that change, (2) runs a subset of security analyses that need to be performed, and (3) updates the security posture of the system by either adding new potential security attacks or removing existing threats. After that, *Policy Synthesizer* takes the analysis results computed by the previous component and constructs security policies in the form of Event-Condition-Action (*ECA*) rules—which the *Policy Enforcer* enforces at runtime, through various effectors.

5.1 Model Extractor & Synchronizer (MES)

Similar to other self-* software systems, SALMA leverages an abstract representation of the software to manage and adapt the system at runtime. Prior research assumes these models are developed in advance. Given the rich app ecosystem of Android, this assumption does not hold, since users can install a variety of apps that are unknown a priori. To address this challenge, *MES* utilizes static and dynamic analysis techniques to automatically obtain and maintain a precise model of an Android system.

To obtain an architectural model of an Android system, *MES* uses *APKtool* [3], a reverse-engineering tool for Android APK files, to recover an app’s manifest file. *MES* then parses the file to extract the app’s components, their properties, their provided interfaces, the required permissions, and the defined permissions, if any.

Parsing the manifest file is not enough to obtain a system’s architecture, since a large amount of information is latent in the app’s bytecode—including all ICCs, programmatically registered components, or defined interfaces. ICC communications are facilitated either by (1) sending Intents or (2) using URIs (see Section 2). To obtain this information, *MES* utilizes IC3 [53], a tool that extracts Intents and URIs along with their information from apps’ bytecode.

MES determines the permissions enforced by components. In Android, a component enforces permissions to either (1) protect

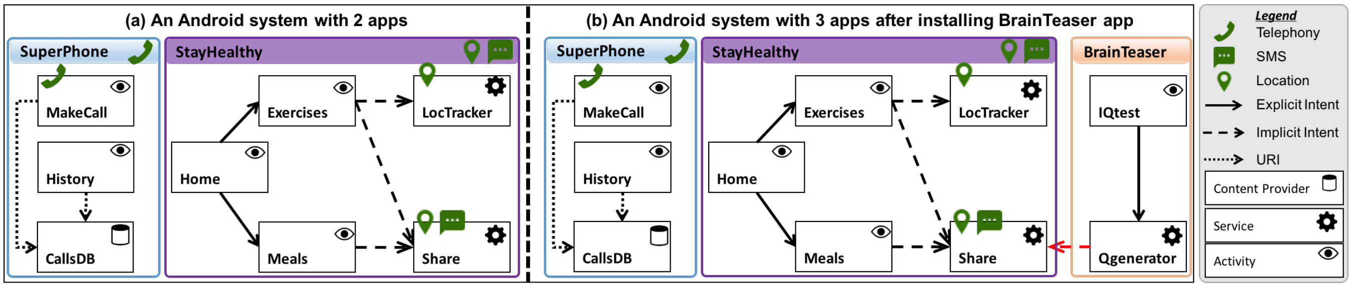


Figure 1: An evolving Android system before, (a), and after, (b), installing `BrainTeaser` app.

access to the entire component or (2) restrict access to parts of the component using permission checks in the code (e.g., use of the `checkCallingPermission` API). *MES* extracts protected access of components from the manifest file and restricted access to parts of components from code. *MES* leverages prior work of architectural modeling [63] to extract enforced permissions from an app's bytecode. Finally, *MES* determines permissions actually used by components either to (1) access a protected Content Provider or (2) call a protected API. To determine such permissions, regarding to (1), we rely on [38] that maps a protected Content Provider to the required permission to access that Content Provider. Regarding to (2), we leverage PScout [16] which maps each sensitive API in Android with the required permission to call that API.

MES captures the architecture of an Android system as a *Multiple-Domain-Matrix (MDM)*[49], which is a matrix representation of all relationship types (i.e., domains) among principal elements, such as components and permissions, in a system. An MDM consists of multiple *Design-Structured Matrices (DSMs)* [65]. Each domain is modeled as a DSM—a simple matrix that captures the relationships of one type. For the purpose of security analysis, SALMA models an Android system using seven domains, four component interaction domains and three permission domains. As a concrete example, Figure 3 shows the derived MDM of the example illustrated in Figure 1 (a). To keep the MDM valid as the system changes, MES synchronizes the MDM with the running system.

The four component interaction domains in the MDM model of Figure 3 represent the various component-to-component communications. Each non-empty cell in these domains indicates that there is a communication between two components, either by sending Intents or using URIs to access the encapsulated data in Content Providers. Rows represent sender components; columns represent receiver components. The *explicit* and the *implicit* communication domains show all component-to-component interactions using explicit and implicit Intents, respectively. Similarly, the data *access* and the data *manipulation* domains show component-to-content provider interactions for reading (i.e., querying) and modifying (i.e., updating, inserting, or deleting) stored data, respectively.

The three permission domains in the MDM model of Figure 3 represent the various component-to-permission relationships. The permission *usage* domain shows that a permission is making protected API calls. The permission *granted* domain shows that a permission is granted to a component T because it is either (1) directly using the permission; or (2) its parent app requests that permission, and T is interacting with another component that uses that permission. The permission *enforcement* domain shows that a permission is enforced by a component through its manifest file or in its code.

To derive the MDM representation of an Android system, *MES* asynchronously interacts with the *Static Analysis Engine* (see Figure 2), a cloud-based web service that leverages various static analysis tools [3, 52, 53, 63] to extract a model of an Android system. Once the *Static Analysis Engine* analyzes the requested app(s), it returns a tuple $S = (C, I, DBR, DBW, P)$ to the *MES*. In the tuple S, C is a set of components; I is a set of Intents; DBR is a set of database read accesses; DBW is a set of database write requests; and P is a set of permissions. Using this extracted information (i.e., tuple S), and SALMA's knowledge of the Android framework embodied in a set of definitions, formally presented below, it derives the MDM representation of an Android system. The *explicit* communication domain is derived using the following rule.

Definition 1 (Explicit Communication). *Let c_1 and c_2 be two arbitrary components in the system, i.e., $\{c_1, c_2\} \subseteq C$, i be an Intent, i.e., $i \in I$. We say that c_1 can explicitly communicate with c_2, if i is sent by c_1, i.e., $i.sender = c_1$, and c_2 is explicitly specified in the Intent i as a target component, i.e., $i.target = c_2$, and either both c_1 and c_2 belong to the same app or c_1 is granted the permissions enforced by c_2:*

$$communicate_e(c_1, c_2) \equiv \exists i \in I \mid i.sender = c_1 \wedge i.target = c_2 \wedge (app_{c_1} = app_{c_2} \vee enforced_{c_2} \subseteq granted_{c_1})$$

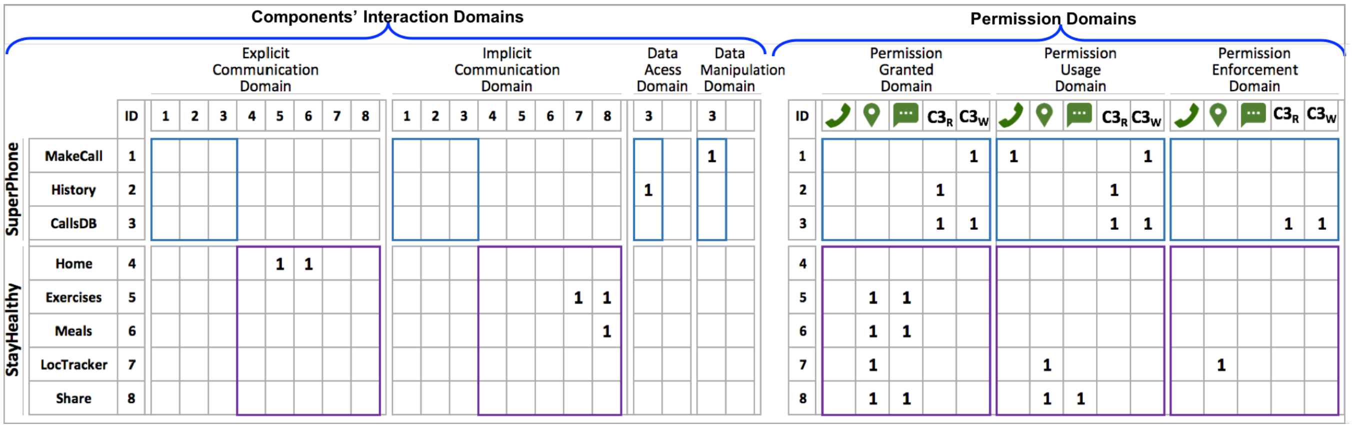
The *explicit* communication domain in Figure 3 shows the results of applying definition 1 to all extracted Intents, i.e., the set I. According to definition 1, component 4 explicitly communicates with component 5, since there is an explicit Intent sent by `Home` to `Exercises` (recall Figure 1 (a)).

Similarly, the communications in the *implicit* communication domain are derived using the following rule.

Definition 2 (Implicit Communication). *Let c_1 and c_2 be two arbitrary components in the system, i.e., $\{c_1, c_2\} \subseteq C$, i be an Intent, i.e., $i \in I$. We say that c_1 can communicate with c_2, if i is sent by c_1, i.e., $i.sender = c_1$, and c_2 is exporting an Intent Filter that can handle i, i.e., $match(i, c_2.f)$, and either both c_1 and c_2 belong to the same app or c_1 is granted the permissions enforced by c_2:*

$$communicate_i(c_1, c_2) \equiv \exists i \in I \mid i.sender = c_1 \wedge match(i, c_2.f) \wedge (app_{c_2} = app_{c_1} \vee enforced_{c_2} \subseteq granted_{c_1})$$

The $match(i, c_2.f)$ function in Definition 2 performs Intent resolution [7] to check if there is an *Intent Filter* declared by c_2 that can handle the Intent i. The *implicit* communication domain in Figure 3 shows the results of applying Definition 2 to all extracted Intents, i.e., the set I. According to Definition 2, component 5 implicitly communicates with component 7 since there is an implicit Intent sent by `Exercises` in which `LocTracker` can handle (recall Figure 1 (a)).



			Components' Interaction Domains																			
			Explicit Communication Domain								Implicit Communication Domain								Data Acess Domain		Data Manipulation Domain	
		ID	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	3		3	
SuperPhone	MakeCall	1																			1	
	History	2																	1			
	CallsDB	3																				
StayHealthy	Home	4					1	1														
	Exercises	5															1	1				
	Meals	6																1				
	LocTracker	7																				
	Share	8																				

			Permission Domains														
			Permission Granted Domain					Permission Usage Domain					Permission Enforcement Domain				
		ID	☎	📍	💬	$C3_R$	$C3_W$	☎	📍	💬	$C3_R$	$C3_W$	☎	📍	💬	$C3_R$	$C3_W$
SuperPhone	MakeCall	1					1	1				1					
	History	2				1					1						
	CallsDB	3				1	1				1	1				1	1
StayHealthy	Home	4															
	Exercises	5		1	1												
	Meals	6		1	1												
	LocTracker	7		1					1					1			
	Share	8		1	1				1	1							

Figure 3: An MDM representation of the system illustrated in Figure 1(a). Each colored box in the MDM corresponds to the matching colored app in Figure 1(a).

The *data access* and the *data manipulation* domains are derived using the following rules.

Definition 3 (Data Access). *Let c be an arbitrary component in the system, i.e., c ∈ C, cp be a content provider, i.e., cp ∈ C, and dbr be a database read (query) request in the system, i.e., dbr ∈ DBR. We say that c can access the stored data in cp, if c sends a database query (dbr) where the authority attribute of dbr matches the authority name of cp, and either c and cp belong to the same app or c is granted the enforced read access permission by cp.*

$$access(c, cp) \equiv \exists dbr \in DBR \mid dbr.sender = c \land dbr.authority = cp.authority \land (app_c = app_{cp} \lor read_{cp} \subseteq granted_c)$$

To illustrate an instance of Definition 3 on the extracted database requests, i.e., *DBR*, Figure 1 shows that component 2 accesses the stored data in component 3 —which is also reflected in Figure 3.

Definition 4 (Data Manipulation). *Let c be an arbitrary component in the system, i.e., c ∈ C, cp be a content provider, i.e., cp ∈ C, and dbw be a database write (insert, delete, or update) request in the system, i.e., dbw ∈ DBW. We say that c can access the stored data in cp, if c sends a database manipulation request (dbw) where the authority attribute of dbw matches the authority name of cp, and either c and cp belong to the same app or c is granted the enforced write access permission by cp.*

$$manipulate(c, cp) \equiv \exists dbw \in DBW \mid dbw.sender = c \land dbw.authority = cp.authority \land (app_c = app_{cp} \lor write_{cp} \subseteq granted_c)$$

As an example of Definition 4, Figure 1 depicts component 1, `MakesCall`, updates the stored data in component 3, `CallsDB`—which is further shown in Figure 3.

Table 1 shows a list of the events (i.e., changes in the system) that *MES* tracks. In this paper, we refer to these events as *significant events.* For each significant event, *MES* receives a notification from the system and updates the model accordingly. For example, when a user installs a new app, *MES* receives a system notification with the ACTION_PACKAGE_ADDED Intent action. In this case, *MES* obtains the architecture of the new app, i.e., $M = (C, I, DBR, DBW, P)$, from the *Static Analysis Engine*; merges *M* with *S*; and applies Definitions 1–4 to add the new app to the current MDM. To avoid substantial analysis time caused by running static analysis tools, the *Static Analysis Engine* can analyze Android apps in advance without waiting for a user to install an app.

Table 1: The Significant Events that SALMA Monitors.

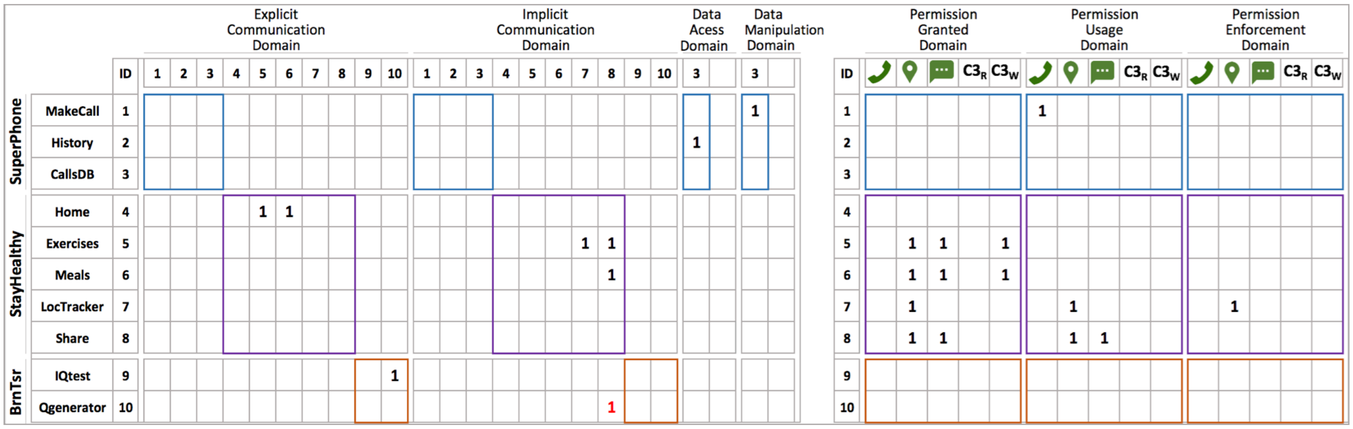
ID	Event	ID	Event
1	ADD_APP	5	NEW_IMPLICIT_COMM
2	REMOVE_APP	6	NEW_EXPLICIT_COMM
3	GRANT_PERMISSION	7	NEW_DATA_ACCESS
4	REVOKE_PERMISSION	8	NEW_DATA_MANIPULATION

In our running example, after a user installs the `BrainTeaser` app (see Figure 1(b)), *MES* updates the maintained model. The MDM illustrated in Figure 4 displays the results of merging `BrainTeaser` with the current MDM, presented in Figure 3. Figure 4 shows that `IQtest` explicitly communicates with `Qgenerator`, which implicitly communicates with `Share` (see Figure 1(b)).

To synchronize the runtime model with the system, *MES* relies on receiving system notifications from the sensors in Figure 2, indicating significant changes that occur in the system. Some system notifications are already implemented in the Android framework, such as `ADD_APP` and `REMOVE_APP` events. For these events, the framework sends broadcast Intents with `ACTION_PACKAGE_ADDED` and `ACTION_PACKAGE_REMOVED` actions, respectively. While in all other significant events, see Table 1, the framework silently executes the event. Therefore, we have introduced new system-generated broadcast Intents to the Android platform. The new broadcasts inform *MES* of certain events whenever they occur in the system. Each system-generated broadcast Intent contains information about a particular event. For example, in case a user grants a permission to an app, the framework sends a `GRANT_PERMISSION` broadcast Intent with the permission name and the application package name.

5.2 Incremental Security Analyzer (ISA)

Android systems are highly dynamic software systems. Reanalyzing the entire system every time a change occurs is neither efficient nor scalable. Therefore, our approach incrementally analyzes the system whenever a change in the system occurs. Our approach leverages the fact that a change in the system (1) impacts only part



			Explicit Communication Domain										Implicit Communication Domain										Data Acess Domain		Data Manipulation Domain	
		ID	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	3		3	
SuperPhone	MakeCall	1																							1	
	History	2																					1			
	CallsDB	3																								
StayHealthy	Home	4					1	1																		
	Exercises	5																	1	1						
	Meals	6																		1						
	LocTracker	7																								
	Share	8																								
BrnTsr	IQtest	9										1														
	Qgenerator	10																		1						

	Permission Granted Domain					Permission Usage Domain					Permission Enforcement Domain				
ID	☎	📍	💬	$C3_R$	$C3_W$	☎	📍	💬	$C3_R$	$C3_W$	☎	📍	💬	$C3_R$	$C3_W$
1						1									
2															
3															
4															
5		1	1		1										
6		1	1		1										
7		1					1					1			
8		1	1				1	1							
9															
10															

Figure 4: An MDM representation of the system illustrated in Figure 1(b) . Each colored box in the MDM corresponds to the matching colored app in Figure 1(b).

of the system and (2) often requires running a subset of the security analyses on the impacted part of the system. In Section 5.2.1, we describe how our approach determines the impacted parts of the system after a change occurs. Section 5.2.2 describes the security rules that our approach applies on the impacted parts of the system to detect the potential security attacks presented in Section 2.

5.2.1 Change Impact Analysis. This analysis consists of two steps: (1) determining the impacted parts of the MDM and (2) identifying the subset of the security analysis rules, formally specified in Section 5.2.2, that need to be considered. More specifically, in step (1), *ISA* determines the affected parts of the system by calculating $\Delta MDM_e = MDM_{t2} - MDM_{t1}$, where $t2$ is a time after an event e and $t1$ is a time before e.

Each cell in ΔMDM has a value of either −1, 0, or 1. −1 means a relationship in the previous system has been removed after e, e.g., e is the revocation of a permission. 0 means there is no change in that relationship before and after e. 1 indicates that a new relationship is introduced due to e. For example, e may be the introduction of a new communication between two components appearing at runtime due to installing a new app, updating an existing app, dynamically loaded code, or execution of obfuscated code. From the affected relationships, *ISA* determines the impacted domains. Applying ΔMDM to our running system, described in Section 1, reveals that the communications in rows 9 and 10 have been added to the system which belong to the *explicit* and the *implicit* communication domains.

In step (2), *ISA* determines the subset of the security rules that need to be considered in light of the affected domains. To that end, *ISA* uses Table 2, which is a lookup table that maps each *Security Analysis* with the *Involved Domains* in that analysis. This table also shows the security analyses that need to be performed when a specific domain changes. For example, if the *EXPLICIT* domain changes, then *ISA* needs to perform only 3 security analyses instead of all 6 analyses. In our running example where only the *explicit* and the *implicit* domains have been changed after installing `BrainTeaser`, Table 2 indicates that the security posture of the system should be checked against the following security attacks: *Intent Spoofing*, *Unauthorized Intent Receipt*, *Privilege Escalation*, and *Identical Custom Permission*.

Table 2: Security analyses lookup table.

Security Analysis	Involved Domain(s)	
Intent Spoofing	Explicit	Implicit
Unauthorized Intent Receipt	Implicit	
Privilege Escalation	Explicit Granted	Implicit Usage
	Enforcement	
Identical Custom Permission	Explicit Granted	Implicit Enforcement
Passive Data Leak	Data Access	Read Permission
Content Pollution	Data Manipulation	Write Permission

5.2.2 Security Rules. This section describes the security rules that SALMA applies on the impacted parts of the system. Each rule when applied on an interaction between two components would reveal if that interaction is vulnerable to a given security attack.

Security Rule 1 (Unauthorized Intent Receipt). *Let c_m be a malicious component, c_v be a vulnerable component, and c_x be a component that c_v intends to send an implicit Intent i to. c_v and c_x belong to the same app, and c_x declares a provided interface, i.e., an Intent filter, through which c_v aims to communicate with c_x using i. In an unauthorized Intent receipt, c_m can intercept i from c_v by declaring a provided interface similar to the one declared by c_x. As such, c_m may gain access to all enclosed data in any matching Intents meant to be received by c_x.*

$$\exists\, communicate_i(c_v, c_m) \mid (app_{c_v} \neq app_{c_m}) \wedge \exists\, communicate_i(c_v, c_x) \wedge (app_{c_v} = app_{c_x})$$

Security Rule 2 (Intent Spoofing). *Let c_m be a malicious component, c_v be a vulnerable component, and c_x be a component intending to communicate with c_v. c_v and c_x belong to the same app. c_v declares a provided interface, i.e., an Intent filter, through which it aims to communicate with c_x. In Intent Spoofing, c_m can send an Intent to c_v over the Intent filter and force c_v to perform a nefarious action upon receipt of the Intent.*

$$\exists\, (communicate_e(c_m, c_v) \vee communicate_i(c_m, c_v)) \mid (app_{c_v} \neq app_{c_m}) \wedge \exists\, communicate_i(c_x, c_v) \wedge (app_{cv} = app_{cx})$$

Security Rule 3 (Privilege Escalation). *Let p be a permission, c_m be a malicious component that is not granted p, and c_v be a vulnerable component that is granted and uses p but does not enforce the use of p as requested by other components. In privilege escalation, c_m is able to indirectly obtain p by interacting with c_v.*

$$\exists\ (communicate_e(c_m, c_v) \lor communicate_i(c_m, c_v)) \mid p \in used(c_v) \land p \notin granted(c_m) \land p \notin enforced(c_v)$$

Security Rule 4 (Identical Custom Permission). *Let p_m be a custom permission defined by malicious app app_m, i.e., $p_m.definedBy = app_m$, and p_v be a custom permission defined by the vulnerable app app_v, i.e., $p_v.definedBy = app_v$. Both p_v and p_m have the same permission name, i.e., $p_v.name = p_m.name$. c_m is a malicious component in app_m that is granted p_m. c_v is a vulnerable component in app_v that enforces p_v. In an identical custom permission, c_m can communicate with c_v since $p_v.name = p_m.name$, even if p_v and p_m are semantically different permissions.*

$$\exists\ (communicate_e(c_m, c_v) \lor communicate_i(c_m, c_v) \lor access(c_m, c_v) \lor manipulate(c_m, c_v)) \mid app_{c_m} \neq app_{c_v} \land p_m \in granted(c_m) \land p_v \in enforced(c_v) \land p_v.name = p_m.name \land p_m.definedBy \neq p_v.definedBy$$

Security Rule 5 (Passive Data Leak). *Let cp_v be a vulnerable* `Content Provider` *that does not enforce a read access permission, c_m be a malicious component that accesses (queries) the stored data in cp_v. In a passive data leak, c_m can passively disclose the sensitive data stored in cp_v.*

$$\exists\ access(c_m, cp_v) \mid enforce_r(cp_v) = \emptyset$$

In Definition 5, $enforce_r(cp_v)$ refers to the read access permission enforced by the `Content Provider` cp_v. In our approach, for each `Content Provider` component, we add two columns in the permission domains of the MDM: one for the read access permission and the other one for the write access permission. For example, each permission domain in the MDM illustrated in Figure 4 contains two permissions for the `CallsDB` component, $C3_R$ for the read permission and $C3_W$ for the write permission.

Security Rule 6 (Content Pollution). *Let cp_v be a vulnerable* `Content Provider` *that does not enforce a write access permission, c_m be a malicious component that changes (inserts, updates, or deletes) the stored data in cp_v. In the content pollution attack, c_m can inappropriately manipulate the sensitive data stored in cp_v.*

$$\exists manipulate(c_m, cp_v) \mid enforce_w(cp_v) = \emptyset$$

In Definition 6, $enforce_w(cp_v)$ refers to the write access permission enforced by the `Content Provider` cp_v.

Regarding our example, SALMA determines that the security rules 1, 2, 3, and 4 should be applied to all interactions in rows and columns 9 and 10. Running these rules, mainly rule 3 on the communication between `Qgenerator` and `Share`, reveals that the implicit communication in row 10 and column 8 of Figure 4 is vulnerable to privilege escalation attack.

5.3 Policy Synthesizer and Policy Enforcer

After *ISA* determines the security vulnerabilities in the system, the *Policy Synthesizer* creates *context-sensitive* security policies to be executed at runtime. The created security policies, in our approach, follow the Event-Condition-Action (ECA) rules paradigm suitable for rapid evaluation as the system executes. Our approach creates ECA rules that, based on a particular system context, will be executed to prevent security threats. More specifically, SALMA creates ECA rules to prevent the communication between the two components that are involved in an identified security vulnerability.

SALMA further tries to minimize the disruption that the security policies may cause. For example, in the case of a privilege escalation attack, SALMA creates a security policy to prevent a vulnerable communication instead of revoking the escalated permission from the vulnerable app, as proposed in [60]. The later solution disrupts all components in the vulnerable app from using that permission which may stop crucial services provided by the disrupted components such as sending text messages or getting driving directions.

As a concrete example, since the communication between Qgenerator and `Share` is marked as potential privilege escalation attack, SALMA creates the following ECA rule.

Event: $i \in ICC$ occurs
Condition: $i.senderPkg$ = `BrainTeaser` $\land$ $i.senderComp$ = `Qgenerator` $\land$ $i.receiverPkg$ = `StayHealthy` $\land$ $i.receiverComp$ = `Share`
Action: *prevent*

Policy Enforcer administers security policies at runtime through various effectors that we have added to the Android runtime environment, as shown in Figure 2. *Policy Enforcer* applies security policies by intercepting the ICCs (both the Intent-based and the URI-based communications) and the resource access transactions to check if they are allowed or not. For Intent-based communication, *Policy Enforcer* can prevent or allow transactions. For the URI-based ICC transactions, *Policy Enforcer* can prevent a component from accessing or manipulating either (1) the entire `Content Provider` specified in the URI or (2) a specific table or file in that `Content Provider`.

6 IMPLEMENTATION

We have implemented SALMA and its constituent components for our experiments and make it available online for reproducibility and reuse purposes [11]. To keep an Android system's architecture synchronized with the running system, the *Static Analysis Engine* is implemented as a cloud-based web service that leverages several prior static analysis tools [3, 16, 52, 53, 63]. Each tool provides specific information that SALMA uses to tailor the architecture of the system.

MES, *Policy Synthesizer*, and *Policy Enforcer* are implemented on top of the Android Open-Source Project (AOSP) [2] version 6 (Marshmallow), API level 23. AOSP is the open-source repository for the Android system. The enforcement mechanism introduced a new package in the Android runtime environment. We also modified other components such as *ActivityManager*, *ContextWrapper*, *ContentProvider*, and *PackageManager*. The total framework changes account for approximately 600 LOC. These changes allow any existing Android app using version 6 and below to run in our version of the Android runtime environment without modification. We have successfully installed the modified Android system image on a Nexus 5X phone and an Android emulator using Android Fastboot tools [5] and Android debug bridge [1].

7 EXPERIMENTAL EVALUATION

Our evaluation addresses the following research questions:
RQ1. How efficient is SALMA at incrementally analyzing the security posture of Android systems compared to a complete analysis approach?

RQ2 How effective is SALMA at reducing the unnecessary disruption caused by the enforcement of security policies to prevent permission-induced ICC attacks?
RQ3. How effective is SALMA at detecting and preventing security attacks in real-world apps?

7.1 RQ1: Efficiency

For this experiment, we downloaded 984 apps comprising three datasets: a dataset of 370 benign apps, randomly selected from Google Play [6];a dataset of 389 vulnerable apps identified in prior literature [48]; and a dataset of 225 malicious apps obtained from various malware repositories [4, 51, 72].

To measure the efficiency of SALMA's incremental analysis, we compared the performance of SALMA with DELDroid [37, 38], a prior approach that similar to our work analyzes the architecture of an Android system for ICC vulnerabilities and enforces the determined architecture at runtime. However, unlike SALMA, DELDroid is not capable of continuous monitoring and incremental analysis of an evolving Android system. We ran our experiments on a MacBook Pro with 2.2 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM. We repeated our experiments 10 times to achieve a 95% confidence interval.

Figure 5 contains box-and-whisker plots comparing the analysis time of each approach as Android apps are added to the system. We started with an Android system of 120 apps and added one app at a time until the system contained 150 apps. We randomly selected 120 apps from the benign dataset, 15 apps from the vulnerable dataset, and 15 apps from the malicious dataset.

Every time an app is added to the system, SALMA incrementally analyzes the system whereas DELDroid reanalyzes the entire system. As illustrated in Figure 5, the analysis time of SALMA takes, on average, 2 seconds to incrementally analyze an Android system whenever a new app is installed. On the other hand, DELDroid takes, on average, 75 seconds.

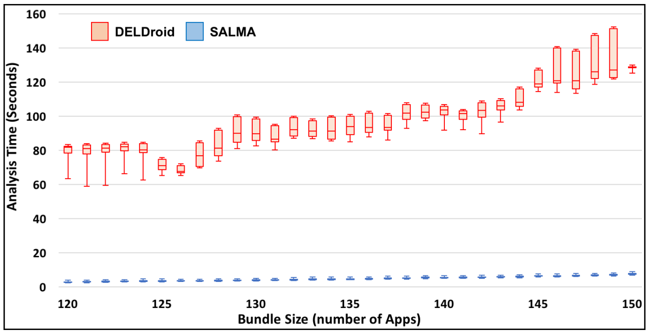


Figure 5: The analysis time of SALMA and DELDroid as Android apps are added to the system.

Figure 6 compares the analysis time of each approach with a decreasing number of apps. We started with a bundle of 150 apps, then we removed one app at a time until the system contained 120 apps. The average analysis time of SALMA is 0.2 seconds while the average analysis time of DELDroid is 35.3 seconds. Due to space limitations, Figures 5 and 6 show the analysis results of adding/removing 30 apps, however, the project's website [11] contains the analysis results of an experiment of adding/removing 80 apps.

Due to the use of code obfuscation and dynamic class loading in Android apps, not all communications can be discovered using static analysis tools. As a result, some communication appears only at runtime, e.g., a new explicit or implicit communication. In such scenarios, SALMA also incrementally reanalyzes the security posture of the system to determine if the new communication poses any threat to the system. If so, SALMA prevents the new communication. In addition to ADD_APP and REMOVE_APP, we assessed the efficiency of SALMA with respect to other system events mentioned in Table 1. We found that SALMA takes, on average across all events, 1.6 milliseconds while DELDroid takes, on average across all events, 63.8 seconds. Due to space limitations, details of this experiment and the evaluation results are reported at the project's website [11].

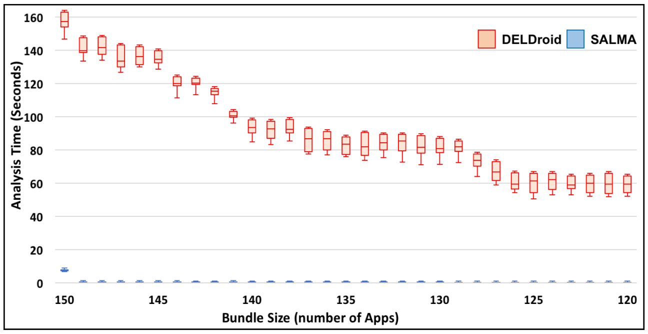


Figure 6: The analysis time of SALMA and DELDroid as Android apps are removed from the system.

SALMA takes about two hours to statically analyze an Android system with 50 apps and determine its initial architectural representation. Thereafter, SALMA incrementally determines the architecture of the running system. Table 3 shows the performance of SALMA in statically analyzing apps, merging/removing an app to/from the architectural model, and intercepting and checking an ICC transaction against the stored security policies. To further improve efficiency, the static analysis time can be performed in advanced without waiting for a user to install an app. All other times in Table 3 cannot be perceived as delays by users, which follows recommendations from Android development guidelines [8].

Table 3: SALMA's static analysis and runtime performance.

	Static analysis (minute)	Merge app to the model (second)	Remove app from the model (second)	Validating ICC trans. (second)
Average	2.3	0.024	0.026	0.025
Std Dev.	1.2	0.027	0.028	0.001

Overall, these results corroborate the efficiency and the scalability of SALMA in incrementally analyzing Android systems.

7.2 RQ2: Disruption

Enforcing security policies at runtime by preventing permission-induced ICC attacks may disrupt benign behaviors of an app. Permission-induced attacks are security breaches enabled by permission misuse, i.e., privilege escalation, identical custom permissions, content pollution, and passive data leaks. Preventing permission-induced attacks can be applied at install-time or runtime [32].

Install-time approaches, such as Kirin [31], prevent the installation of vulnerable apps. Runtime prevention approaches can either (1) prevent only the malicious communication whenever it occurs, as performed in SALMA, DELDroid [38], SEALANT [47], and SEPAR [20]; or (2) revoke permissions of vulnerable apps at runtime, as in TERMINATOR [60] and AppGuard [17].

For this experiment, we analyzed a bundle of 150 apps, the apps used in RQ1, and found that 40 apps are vulnerable to various permission-induced attacks. We then computed the *disruption* in each vulnerable app caused by the enforcement of the various security policy mechanisms discussed earlier. Disruption of an app *a* is computed using the following equation:

$$disruption(a) = \frac{|comps_{disr}(a)|}{|comps_{tot}(a)|} \times 100$$

Where $comps_{disr}(a)$ is the set of components in app a that are disrupted and $comps_{tot}(a)$ are the set of all components in app a. We consider a component c to be disrupted if c uses a permission p involved in a permission-induced attack, since c will be unable to provide its full services if p is revoked.

As an example, consider an app a_v with 5 components where 3 of its components use permission p to provide their services. One component using p is vulnerable to a privilege-escalation attack. In this case, to protect the user, the install-time approaches prevent the installation of a_v, disrupting all of its components, i.e., $disruption(a_v) = 100\%$. On the other hand, approaches that revoke permission will revoke p to prevent the attack, resulting in 60% disruption of that app, i.e., 3 components will not be able to provide their full services due to the lack of the required permission p. However, SALMA, which only prevents malicious communication when it occurs, results in 0% disruption, since all components will be able to provide their full services while keeping the system protected.

Figure 7 compares the three different permission-induced prevention mechanisms. The diagram shows that SALMA has 0.4% disruption, meaning that SALMA does not disturb components from providing their services except in one identical custom permission case. In that case, SALMA created a security policy to revoke a custom permission from the malicious app so it will not be able to access the vulnerable app. On the other hand, the install-time approach performs the worst (100%), as it does not allow installation of a vulnerable app. Finally, revoking permissions at runtime to prevent permission-induced attacks would result, on average per app, in 32% disruption. Meaning that, on average, 32% of the components in a vulnerable app will not be able to provide their full services due to the lack of required permissions even though some of these components are not vulnerable or involved in any vulnerability. Moreover, revoking permissions from apps at runtime lead to crashes or unexpected behaviors due to inappropriate handling of dynamic permissions in Android [61].

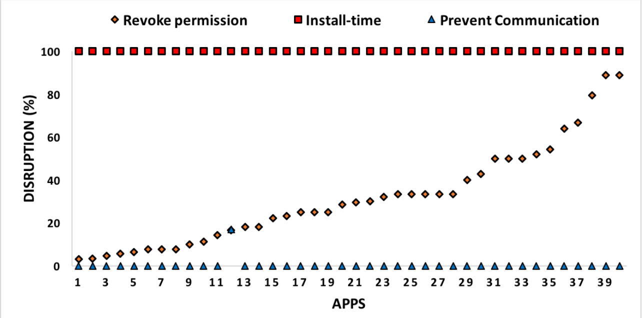


Figure 7: Disruption results for each app

DELDroid, SEPAR, SEALANT, and SALMA all attempt to prevent malicious communication whenever it occurs. However, unlike SALMA, the other three approaches assume that all permissions are granted to all apps indefinitely. This assumption increases those approaches false positives which, in turn, increases unnecessary disruption. For example, a privilege-escalation vulnerability is not exploitable unless the escalated permission is granted to the vulnerable app. However, the three approaches prevent vulnerable communication at all times, while SALMA prevents vulnerable communication only when the system is at risk, i.e., the permission is granted to the vulnerable app.

7.3 RQ3: Attack Detection and Prevention

To evaluate SALMA's ability to detect and prevent security threats, we conducted a thorough evaluation using malicious and vulnerable real-world apps with known security attacks, and compared the detection and prevention results of SALMA with state-of-the-art approaches. We included state-of-the-art approaches that are (1) publically available, (2) provide detection and prevention mechanisms, and (3) extend the Android framework. To that end, we included DELDroid [37, 38], SEPAR [20], and SEALANT [47]. DELDroid determines the least-privilege architecture of an Android system and enforces it at runtime. SEPAR provides an automatic scheme for formal synthesis and enforcement of Android ICC security policies. SEALANT is a technique that combine static analysis with dynamic monitoring to detect security vulnerabilities in Android apps and prevent ICC attacks.

To conduct this experiment, we used 188 malicious and vulnerable apps for which the steps and inputs required to create the attacks were known and documented. These apps have been used in the evaluation of the included approaches. In total, the subject apps contain 94 ICC attacks where 45 of them are hidden attacks, i.e., the malicious code is not part of the apps' bytescode but instead is loaded at runtime using the dynamic class loading feature as described in [57], and the rest (49 attacks) are not hidden attacks, i.e., the malicious code is part of the apps' bytecode. We ran each approach on the subject apps, then deployed the apps on the Android environment, and manually exercised all known attacks. We report the number of detected and prevented attacks for each approach.

The *Attack Detection* column in Table 4 show the evaluation results of each approach for detecting the security attacks. For example, SALMA and DELDroid detected all of the 20 privilege-escalation instances that are not dynamically loaded, i.e., not hidden attacks, whereas SEPAR and SEALANT detected only 12 and 14 attacks, respectively. According to Table 4, SALMA is able to detect all 94 attacks, including the hidden attacks, with no false positives or false negatives, while the detection rate of the other approaches ranges from 16% to 30%. Given the reliance of the included approaches on static program analysis to detect security risks, all of them are unable to detect hidden attacks launched via dynamically loaded code, see the gray area in Table 4. However, since SALMA incrementally analyzes the security posture of the system in response to system changes, i.e., new inter-app communications added at runtime (recall Section 7.1), SALMA is able to detect these attacks at runtime.

The *Attack Prevention* column in Table 4 shows the evaluation results of each approach for thwarting the security attacks at runtime.

Table 4: The ability of SALMA in detecting and preventing security attacks compared to the state-of-the-art approaches.

Attack Type (Count)	Security Attack	# Attacks	**Attack Detection**				**Attack Prevention**			
			DELDroid	SEPAR	SEALANT	**SAlMA**	DELDroid	SEPAR	SEALANT	**SAlMA**
Not hidden (49)	Intent Spoofing	3	3	3	2	3	3	3	2	3
	Unauthorized Intent Receipt	5	5	0	1	5	5	0	1	5
	Privilege Escalation	20	20	12	14	20	20	12	14	20
	Identical Custom Permission	7	0	0	1	7	0	0	1	7
	Content Pollution	7	0	0	0	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7	0	0	0	7
Hidden (45)	Intent Spoofing	13	0	0	0	13	13	0	0	13
	Unauthorized Intent Receipt	2	0	0	0	2	2	0	0	2
	Privilege Escalation	9	0	0	0	9	9	0	0	9
	Identical Custom Permission	7	0	0	0	7	0	0	0	7
	Content Pollution	7	0	0	0	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7	0	0	0	7
Total attacks		94	28	15	18	94	52	15	18	94
Detection / Prevention Rate			30%	16%	19%	100%	55%	16%	19%	100%

SALMA is able to prevent all security attacks in Table 4 at runtime while the prevention rate of the other approaches ranges from 15% to 55%. Interestingly, DELDroid is able to prevent some of the ICC attacks that it did not detect, because it prevents all communications that are not part of the statically determined architecture.

8 RELATED WORK

Attack Detection. Numerous static analysis approaches have been proposed in the literature for detecting ICC attacks in Android systems [48],[68],[46],[50],[19],[70],[23], and [54]. COVERT [19] presents an approach for compositional analysis of Android inter-app vulnerabilities. DidFail [46] tracks data flows between Android components. Similarly, IccTA [48] leverages an Intent resolution analysis to identify inter-component privacy leaks. Unlike SALMA, all of these approaches cannot detect ICC attacks conducted through dynamic class loading.

Attack Prevention. DELDroid [37, 38] is an approach that determines the least-privilege architecture of an Android system and enforces it at runtime. Similar to SALMA, DELDroid analyzes the architecture of an Android system for ICC vulnerabilities and modifies the Android platform to enforce the determined architecture. Unlike SALMA, DELDroid is a design-time solution that (1) does not change the derived architecture as the system evolves; (2) assumes that all permissions are granted to apps indefinitely, which increases disruption; and (3) assumes that all hidden communications are malicious, which further contributes to disruption.

Other approaches, such as [32],[31],[20],[47], and [60], statically analyze Android apps and dynamically enforce security policies to prevent ICC attacks. IPC-Inspection [32] is an OS mechanism for preventing privilege-escalation attacks that reduces the permissions assigned to an app when it communicates with an app having fewer privileges. Kirin [31] detects security vulnerabilities by only analyzing an app's configuration file. SEPAR [20] and SEALANT [47] rely on the analysis results generated by COVERT [19] to prevent ICC attacks at runtime. TERMINATOR [60] performs temporal analysis for preventing permission-induced ICC attacks. All of these tools do not update their models once the system changes.

Another set of approaches leverage dynamic analysis techniques to detect and prevent security attacks [21, 27, 40, 55]. AppFence [40] prevents apps from exfiltration of data outside the device. Quire [27] prevents privilege-escalation attacks from leaking data outside the device using the Internet permission. Saint [55] extends the functionality of Kirin to allow for install-time permission assignment and their run-time use as specified in the policies provided by an app's developer. XManDroid [21] presents a solution for privilege-escalation attacks by restricting communication at runtime between applications that could lead to dangerous information flows. While SALMA automatically analyzes the system and creates security policies to prevent ICC attacks, all of these tools depend upon defining security policies by developers and they require modifications to apps' implementation logic.

9 CONCLUSION

This paper presents SALMA, an automated self-protecting Android system that monitors itself and adapts its behavior at runtime to prevent ICC security risks. SALMA maintains a precise runtime model, represented as a *Multiple-Domain-Matrix (MDM)*, and incrementally and efficiently analyzes an Android system in response to incremental system changes. The maintained architecture is used to reason about the running Android system. Every time the system changes, SALMA determines (1) the impacted part of the system, and (2) the subset of the security analyses that need to be performed, thereby greatly improving the performance of the approach. Our experimental results on hundreds of real-world apps corroborate SALMA's efficiency and scalability with minimal disruption.

Android components are increasingly shipped with native binaries that are shown to have memory-based vulnerabilities (e.g., buffer overflow) [14]. Modeling native code in MDMs, building associated security rules for native-code vulnerabilities, and modeling the interaction among managed and native code in MDMs can provide further attack detection and prevention, but complicate analyses and may lead to scalability issues. Such challenges are interesting avenues of future work.

10 ACKNOWLEDGMENT

We thank Hamid Bagheri for his contributions to this work. 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 FA95 501610030 from the Air Force Office of Scientific Research.

REFERENCES

[1] Android debug bridge. https://developer.android.com/studio/command-line/adb.html, Accessed February 2018.

[2] Android open source project. https://source.android.com/, Accessed February 2018.

[3] Apktool. https://ibotpeaches.github.io/Apktool/, Accessed February 2018.

[4] Contagio malware repository. http://contagiodump.blogspot.it, Accessed February 2018.

[5] Fastboot. https://source.android.com/source/running.html, Accessed February 2018.

[6] Google play. https://play.google.com/store?hl=en, Accessed February 2018.

[7] Intent resolution. https://developer.android.com/guide/components/intents-filters.html, Accessed February 2018.

[8] Keeping your app responsive. https://developer.android.com/training/articles/perf-anr.html, Accessed February 2018.

[9] Nokia threat intelligence report. https://onestore.nokia.com/asset/200492/Nokia_Threat_Intelligence_1H2016_Report_EN.pdf, Accessed February 2018.

[10] Number of available apps in the google play store. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, Accessed February 2018.

[11] Salma: Self-protection of android systems from inter-component communication attacks. https://www.ics.uci.edu/ seal/projects/salma/index.html, July 2018.

[12] Smartphone os market share, 2017 q1. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, Accessed February 2018.

[13] So many apps, so much more time for entertainment. http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html, Accessed February 2018.

[14] B. Aloraini and M. Nagappan. Evaluating state-of-the-art free and open source static analysis tools against buffer errors in android apps. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 295–306. IEEE, 2017.

[15] S. Arzt, K. Falzon, A. Follner, S. Rasthofer, E. Bodden, and V. Stolz. How useful are existing monitoring languages for securing android apps? In *Software Engineering (Workshops)*, pages 107–122. Citeseer, 2013.

[16] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, pages 217–228, Raleigh, NC, October 2012.

[17] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer, 2013.

[18] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *International Symposium on Formal Methods*, pages 73–89, Oslo, Norway, June 2015. Springer.

[19] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, September 2015.

[20] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 514–525, Toulouse, France, June 2016. IEEE.

[21] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[22] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19. Citeseer, 2012.

[23] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *International Conference on Mobile Systems, Applications, and Services*, pages 239–252, Bethesda, Maryland, June 2011. ACM.

[24] H. Cho, J. Lim, H. Kim, and J. H. Yi. Anti-debugging scheme for protecting mobile apps on android platform. *The Journal of Supercomputing*, 72(1):232–246, 2016.

[25] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *Working Conference on Reverse Engineering*, Washington, DC, October 2009. IEEE.

[26] A. Danielescu. Anti-debugging and anti-emulation techniques. *CodeBreakers Journal*, 5(1), 2008.

[27] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, page 3, 2011.

[28] A. Egners, U. Meyer, and B. Marschollek. Messing with Android's permission model. In *Int'l Conf. on Trust, Security and Privacy in Computing and Communications*, pages 505–514, Liverpool, United Kingdom, June 2012. IEEE.

[29] K. O. Elish, D. Yao, and B. G. Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*, 2015.

[30] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[31] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, Chicago, Illinois, November 2009. ACM.

[32] Z. Fang, W. Han, and Y. Li. Permission based Android security: Issues and countermeasures. *Computers & Security*, 43:205–218, June 2014.

[33] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[34] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, page 88, San Francisco, California, August 2011.

[35] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. *IEEE Security & Privacy*, 5(3), 2007.

[36] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek. Automatic generation of inter-component communication exploits for android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671. ACM, 2017.

[37] M. Hammad, H. Bagheri, and S. Malek. DELDroid: Determination and enforcement of least-privilege architecture in android. *Institute for Software Research, University of California, Irvine. UCI-ISR-18-2*.

[38] M. Hammad, H. Bagheri, and S. Malek. Determination and enforcement of least-privilege architecture in android. In *IEEE International Conference on Software Architecture (ICSA)*, pages 59–68, Gothenburg, Sweden, April 2017. IEEE.

[39] S. Holla and M. M. Katti. Android based mobile application development and its security. *International Journal of Computer Trends and Technology*, 3(3):486–490, 2012.

[40] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. 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*, pages 639–652. ACM, 2011.

[41] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[42] Y. Z. X. Jiang and Z. Xuxian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[43] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.

[44] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 69–80. ACM, 2012.

[45] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[46] W. Klieber et al. Android taint flow analysis for app sets. In *International Workshop on the State of the Art in Java Program Analysis*, Edinburgh, United Kingdom, June 2014. ACM.

[47] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic. A sealant for inter-app security holes in android. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 312–323. IEEE, 2017.

[48] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Int'l Conf. on Software Engineering*, pages 280–291, Florence, Italy, May 2015. IEEE.

[49] U. Lindemann and M. Maurer. Facing multi-domain complexity in product development. In *The future of product development*. Springer, Berlin, Germany, 2007.

[50] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *conference on Computer and communications security*, pages 229–240, New York, NY, October 2012. ACM.

[51] F. Maggi, A. Valdi, and S. Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 49–54, Berlin, Germany, November 2013.

[52] D. Octeau, S. Jha, and P. McDaniel. Retargeting android applications to java bytecode. In *International Symposium on the Foundations of Software Engineering*, page 6. ACM, 2012.

[53] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Int'l Conf. on Software Engineering*, pages 77–88, Florence, Italy, May 2015. IEEE.

[54] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Sec. Symp.*, Washington DC, Aug. 2013.

[55] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[56] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[57] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, San Diego, California, February 2014.

[58] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. A framework for understanding dynamic anti-analysis defenses. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 2. ACM, 2014.

[59] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*, pages 187–202. Springer, 2015.

[60] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek. A temporal permission analysis and enforcement framework for android. In *International Conference of Software Engineering (ICSE '18)*, Gothenburg, Sweden, May 2018. IEEE.

[61] A. Sadeghi, R. Jabbarvand, and S. Malek. Patdroid: permission-aware gui testing of android. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, pages 220–232. ACM, September 2017.

[62] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[63] B. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, and D. Garlan. Architecture modeling and analysis of security in android systems. In *European Conference on Software Architecture*, pages 274–290, Copenhagen, Denmark, November 2016.

[64] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A Small But Non-negligible Flaw in the Android Permission Scheme. In *Int'l Symp. on Policies for Distributed Systems and Networks*, pages 107–110, Fairfax, VA, July 2010.

[65] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE transactions on Engineering Management*, (3):71–74, 1981.

[66] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[67] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[68] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM CCS*, Scottsdale, Arizona, November 2014.

[69] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[70] K. Xu, Y. Li, and R. H. Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[71] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[72] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, San Francisco, California, May 2012. IEEE.