How much is too much?: Untrusted Android Applications & Permission Revolution

Shubham Agarwal
Saarland University
Saarbrücken, Germany
s8shagar@stud.uni-saarland.de

***Abstract*—Smartphone applications have redefined the way people use mobile devices in their everyday lives ever since their inception in the past decade. At the same time, there have been several reported instances, where nefarious smartphone applications exploit the permissions originally granted by the user to carry out intended functionalities to otherwise leak private user information from user's device. Moreover, these malicious applications could also abuse other application installed on the target device with higher privileges to perform unintended security-critical operations, commonly known as application-level privilege-escalation vulnerabilities.**

In this seminar report, we discuss in detail the two well-known works in the early days of Android, focusing on two orthogonal aspects: application-level privacy and security in the Android ecosystem, respectively. We outline the threats faced by the application users, the design issues and other challenges encountered by the security engineers and application developers. We then highlight the methodology and countermeasures to defend against such threats in the wild as proposed by the authors. Based on the above works, we discuss the evolution of the mobile ecosystem, the changes and security enhancements in the Android Permissions Framework and fine-grained control over private data by exploring further research works in this domain. In the end, we discuss the pros and cons of the proposed approaches, the threats and challenges which continue to exist in the ecosystem.

I. Introduction

Today, smartphones are not just viewed as a medium of telephonic conversation, thanks to the multitude of functionalities provided by smartphone applications. These devices are popular to such an extent that over 50% of all web traffic originated from smartphones since 2017, as also reported in [1]. Smartphone applications provide a wide range of capabilities to their end-users such as entertainment, navigation, e-banking, web browsing, online payments, communication and whatnot. Along with pre-installed default applications that provide basic services such as call and messaging, there are third-party applications hosted on common marketplaces which provide to a wide range of services such as video games, payment and banking, audio and video streaming applications, and so on.

Google Android is one of the most popular and widely used mobile operating systems for various mobile platforms. As of September 2020, there are over 3 million applications hosted on the Google Play Store catering services to the Android users. While there also exist other third-part stores where users can download and install applications from, Google Play Store is the de-facto platform which hosts applications for smartphones enabled by the Android OS. While these smartphone applications are popular and useful, they also pose a significant threat to the end-user since they have access to the tremendous amount of personally identifiable information (PII) stored by the users. Numerous third-party applications have been reported in the past to steal sensitive user information such as contacts, call logs, location data, device usage statistics and further send them to the remote server. These malicious operations often occur covertly when the user is not interacting with the device or would hide behind a seemingly expected or benign use-cases.

Furthermore, it may even be possible for these applications to execute undesirable security-critical operations on behalf of the user for which the users have no idea about. It is possible by communicating to other benign or rogue applications with higher privileges installed on the same device and ask them to execute malicious operations for itself, thus, effectively escalating their privileges. For instance, a file manager application, which is originally only permitted to access and manage files on the device, may be able to send them to a remote server by asking another web browsing application which can access the Internet.

Considering the privacy of user information and the security of these third-party applications, it is imperative for the research community to understand the requirements from the users' perspective as well as the challenges faced by the security engineers and developers to tackle against these threats in the wild. In this report, we consider the design solutions proposed in the early days of the Android by the authors in [2] & [3] as our lead study. We further identify a few related follow-up studies which depicts the evolutionary trend of Android Permission Model and discuss the approaches implemented in these works to analyze, detect and defend against malicious third-party applications.

To summarize, the key research contributions to this report are as follows:

- We outline how the Android Permission Framework controls the capabilities of an application installed on the device and briefly discuss a specific class of security vulnerability: privilege escalation attacks.
- We then discuss the *privacy mode* proposed in [2] to enable users to prevent data leaks by untrusted third-party applications.
- We further elaborate on a system-centric, policy-driven solution proposed in [3] to defend against variants of

privilege escalation attacks at application-level.

- In the end, we discuss subsequent related research works to understand the current state of privacy and security issues in the ecosystem.

II. Related Work

We describe a few of the research studies which are closely associated with our lead studies. We consider those works which were published before as well as after the lead works in both the orthogonal directions to understand the trend of the evolution of Android ecosystem, the permission framework as well as challenges faced by different approaches.

Before [2] & [3] by Zhou et al. & Bugiel et al., respectively, researchers proposed a few information flow tracking approaches to detect privacy-violating applications as well as to tackle privilege escalation attacks. TaintDroid [4] uses dynamic taint-tracking method to identify private-information that flows across information and leaves through Network sockets at runtime. Since it only tracks data flows and not control flows, this approach could only be used to detect privacy leaks but not privilege escalations.

Other approaches include analyzing the permissions that the application requests in the manifest and then check at install-time whether the functionality complies with it or breaks any manually defined security rules. For instance, Kirin [5] verifies the requested permission and their operations at install-time against the defined system-centric policy and further allow or deny the access. This approach may result in over-approximation of potential communication links and would lead to high false-positives. SAINT [6] puts an onus on the application-developer to define privacy and security rules which will regulate the applications' runtime behaviour and prevent against confused-deputy attack. Similarly, approaches such as in IPC Inspection [7] aims to defend against confused deputy attacks by IPC provenance and reducing permission to the original callers' ability. However, it does not protect against malicious developers and rogue policies defined by them.

Apex [8] tries to limit the runtime behaviour of applications to explicitly restrict the usage of phone resources by allowing users to selectively grant or deny access at runtime. While these works try to detect the privacy-violating data flows based on different approaches, a user-driven access-control at runtime with fail-safe defaults is necessary to prevent data leaks without affecting the user experience as well as the functionality of benign applications.

With changing permission model and reported privacy leaks, the researchers in their later works put more stress on the context in which the application uses the permission to access private data. Wijesekera et al., in [9], conducts user studies to determine when would they want to allow or deny certain privacy-sensitive resources. Further, Votipka et al., in [10], also tries to distinguish the context of background data access and understand users' choice and comfort if they were aware of such operations. Liu et al., in [11], and Raval et al., in [12], propose an exclusive permission-monitoring application or plugin which would control the behaviour of other applications at runtime. Chitkara et al., in [13], and Diamantaris et al., in [14], the origin of the access request should be distinguished between the untrusted application and the third-party libraries embedded within, and the final decision to allow or deny access should be based on this origin.

Later works targeting privilege escalation attack also focused on the system-centric solution, as also argued by the authors in [3]. Bugiel et al., in [15], proposes a fine-grained per-app access-control model using SELinux policies with additional security modes to suit users' requirements. In line with previous works, Backes et al. further introduces SCIPPA [16], an IPC provenance mechanism at the system level that could be used to prevent confused deputy attack. Further, Elish et al., in [17], conducts large-scale analysis on collusion patterns among Android applications and suggests that a practical solution is required to defend against such attack and, at the same time, do not yield significant false positives. FlexDroid [18] argues for in-application privilege separation using stack inspection between the application and the embedded third-party libraries and prevent any privilege escalation arising from these libraries.

III. Background

In this section, we briefly discuss the technical details to highlight application communication channels, the permission model that restricts their functionality as well as the privilege-escalation vulnerability,

A. Inter-Application Communication Channels

Android application can communicate with each other using Binder-IPC channel provided by the Android middleware. The *Reference Monitor* at the middleware exercise mandatory access-control and authorizes the *Intent*, i.e. the request at the application level, and further allows the communication to proceed. Since this mechanism involves communication between two components at the application level, it also is known as *Inter-Component Communication* (ICC). However, the applications may even communicate via indirect, kernel-level channels, bypassing the MAC at middleware, such as by establishing Internet socket or through the shared filesystem.

B. Android Permission Model

Every Android application contains a *manifest* file which contains the list of permissions that it requires to conduct its expected functionalities after installation. Ideally, the applications must request only those permission from the user which is essential for the intended functionality and nothing more than that, following the *Principle of Least Privilege*. Depending upon the Android version, the user can selectively grant or deny the requested access by the application at install-time or run-time, as well as modify them as per their needs at any time. The permissions are also categorized into different protection levels based on the criticality of the operations: normal, dangerous and signature permissions. However, before Android API 23, the user would need to grant all the permissions requested by the application to complete the installation, or else, the process would abort. Also, once granted, there was no permission revocation by which users could discard initially granted permissions to any application. Since both our lead studies date back to the permission model before API 23,

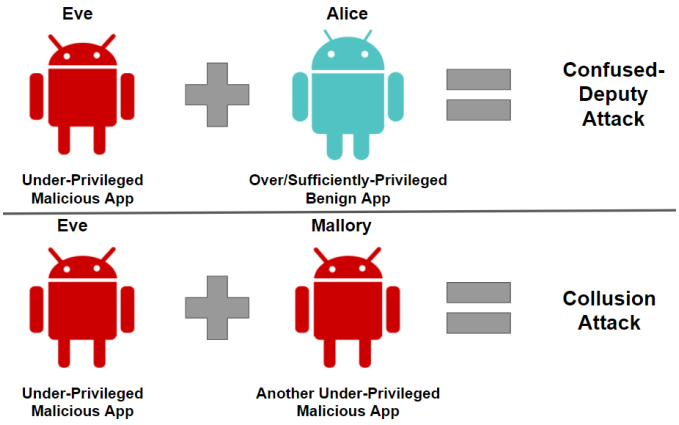

Fig. 1: Privilege Escalation Vulnerabilities

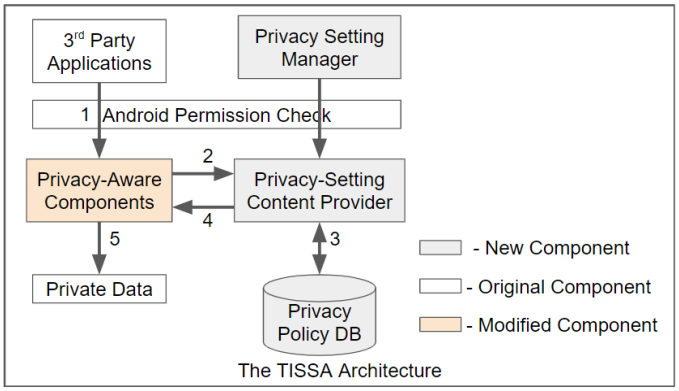

Fig. 2: TISSA Architecture - Privacy Mode at Application Layer

we consider the older model while discussing them in later sections.

Whenever installing an Android application on the device, the Linux kernel assigns a unique identifier *uid* to the application process and isolates it within an application sandbox. It can only access resources that are available within this sandbox or from the shared filesystem. There may exist more than one application within a sandbox and may have a shared UID, given that they are signed with the same developer certificate.

C. Privilege Escalation Vulnerabilities

We understand that an Android application is ideally allowed to perform only those operations for which the users have explicitly granted the permissions. However, privilege escalation is a class of vulnerability where an application could effectively execute operations that require higher privileges and for which the user has not granted the permission. As described in Fig. 1, this could happen in either of the two following ways: (i) when a malicious application either communicates with a benign yet vulnerable application with higher privileges and asks to perform operations for them, often known as confused-deputy attack, or (ii) the application colludes with another malicious application installed on the device fulfil their intent, as in SoundComber attack [19], called collusion attack. Malicious applications may communicate to each other using direct communication channel such as Binder-IPC or may resort to indirect channels such as communication over the Internet, or writing and reading from the shared filesystem, thus bypassing the reference monitor checks completely, thus making it difficult to thwart this vulnerability at the application level.

IV. Privacy Leaks & TISSA

Once the user grants any requested permission to the application, it may use the permission to perform the associated operations even when the user would not want to. There exists no mechanism to control the context in which the application should utilize the granted permissions. Further, there was also no revocation mechanism in place which allowed users to discard the permissions once granted to the applications at install-time. Zhou et al., in [2], proposed an additional permission specification and enforcement mechanism at the application layer, referred to as the *privacy mode*, which aimed to prevent the untrusted third-party applications from leaking any sensitive personal information.

A. Problem Statement

The primary goals of this study are as follows:

- To allow the user to control the data access and flow of private information among untrusted third-party applications at runtime.
- To allow the user to modify the permission configurations for the applications after their installation and without affecting their core functionality.
- The framework should be able to gracefully handle the crashes that may result from modifying permissions at runtime and provide fail-safe defaults for the same.

B. TISSA Framework

The authors propose the design of TISSA framework that would enforce the privacy mode on top of the existing Android security mechanism. We only include the untrusted third-party applications into out threat model while the underlying OS kernel, system services and other pre-installed applications constitute the trusted base for this study. The requirements for this framework are: *a*) the proposed solution should be lightweight in nature and must not affect the usability of the application by elevating its memory and performance overhead in significant terms, *b*) the security mechanism should be backward compatible with the existing Android applications and by no means interfere with their execution, and *c*) minimal changes should be made to the existing framework for better adaptability and portability.

Fig. 2 depicts the complete architecture of the proposed TISSA framework. In this study, we consider the contacts content provider, the telephony manager and the location manager as the privacy-aware components to demonstrate the approach. We include a few additional components in the existing permission model while we also modify the above-mentioned privacy-aware components as part of our proposed

framework. To implement all the proposed components, it involves approximately a thousand LOC in Java.

The *privacy-setting content provider* manages an SQLite database that constitutes the privacy policies configured for each application by the user. This content provider exposes an API such that the privacy-aware components can query the database to get the settings defined by the user and process the incoming requests correspondingly. The *privacy-setting manager* is an Android application that provides an interface to the user to define privacy policies for the third-party applications installed on the device. The underlying Android origin-based sandboxing mechanism and permission model protects the integrity of these newly added components such that both these components are signed by the same certificate. Thus, only privacy-setting manager could be used to set or modify the settings managed by the corresponding content provider.

The *privacy-aware components*, such as the telephony manager or the location manager, are modified such that they first verify the authenticity of the incoming request with the original Android permission model, as usual. When the Android successfully allows the data access request to be processed, these components are modified such that they additionally retrieves the privacy settings configured for the requesting application and stored by the user. It does so by extracting the application information (*package id*) from the request and fetch the settings via the API exposed by the privacy-setting content provider. In case when an application wants to access location data, there exist two possible ways to do so: (i) by directly requesting from the `LocationManager`, or (ii) by registering a `Listener` which would be triggered whenever the device location is updated. The privacy-aware component intercepts the access-request from both the mediums for the sake of completeness.

The framework allows the user to set the privacy policies for any third-party application at different granular levels. An application could be entirely trusted such that it permits access to all the requests. Users can also define a partial trust or second-level policy specification for individual applications where they are allowed to specify desired policies for each possible access-requests from the respective application. One can also totally distrust them and select to restrict access to any private data. It means that though there does not exist any permission revocation mechanism in Android, using the TISSA framework, the user would be able to discard certain permissions without affecting the normal execution of the application or the Android security checks. On an even higher level, another alternate privacy setting proposed by the authors would be a restrictive approach where an untrusted application cannot access any private data from the device and a permissive alternate where the access to any resource by any application would fall back to the original Android permissions assigned by the user at install-time.

While the user can allow the application to access the private data whenever it deems fit to them, they can deny the request in more than one ways. Whenever the user wants to restrict access to the private data for any application, the framework allows them to choose among three possible options: *empty*, *fake* and *anonymized*. While *empty* resembles non-availability of data, *fake* & *anonymized* data could still be used to carry out functionality, depending upon the nature of the data and without revealing any undesired information. However, this fail-safe default would highly depend upon the nature of the application and the functionality for which the data is used.

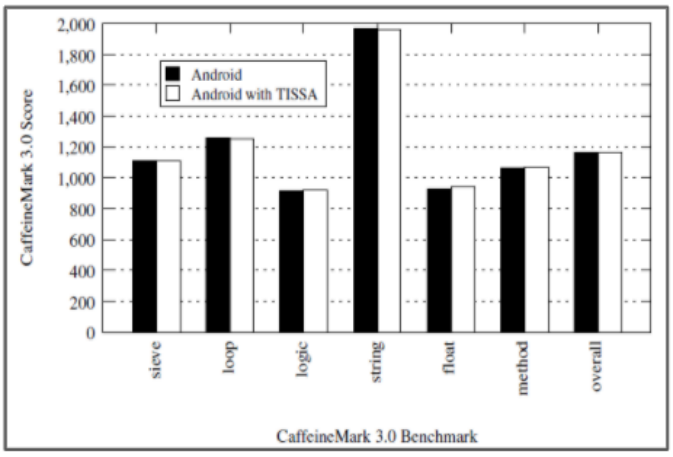

Fig. 3: TISSA - The Result of CaffeineMark 3.0 Benchmark

C. Evaluation & Result

The authors select a total of 24 applications categorized into two distinct sets to evaluate the proposed TISSA framework and its feasibility based on two key parameters: effectiveness and performance. The first set contains 13 untrusted third-party applications which are known to leak private information by TaintDroid, in [4]. The other dataset contains 11 random applications from the Play Store that communicates with at least one of the privacy-aware components chosen for this study and accesses private information. We evaluate the framework with a restrictive approach so that no untrusted applications are allowed to access any confidential information whatsoever.

Effectiveness: While running first set of applications with the restrictive approach, the framework is effectively able to intercept all the access requests to private data and further prevent them from being leaked based on the privacy policies defined for them. From the second set of randomly selected applications, TISSA captured suspicious requests accessing device identity & location information and supplied them with fake data as defined in the privacy policies. The authors verified these suspicious requests with the TaintDroid and confirmed that the applications indeed leak private data to the remote server. Among 24 applications, 14 of them leaked location data & 13 of them leaked device identity information, while 6 of them leaked both.

Performance: In Fig. 3, we observe that TISSA does not incur any significant performance overhead while executing third-party applications and thus, enable user-controlled application-level privacy protection without affecting the usability of the application and kernel-level modifications.

V. Privilege Escalation Attacks & XManDroid*

Based on their requirements, we classify the privilege escalation attack into two categories: confused-deputy attacks

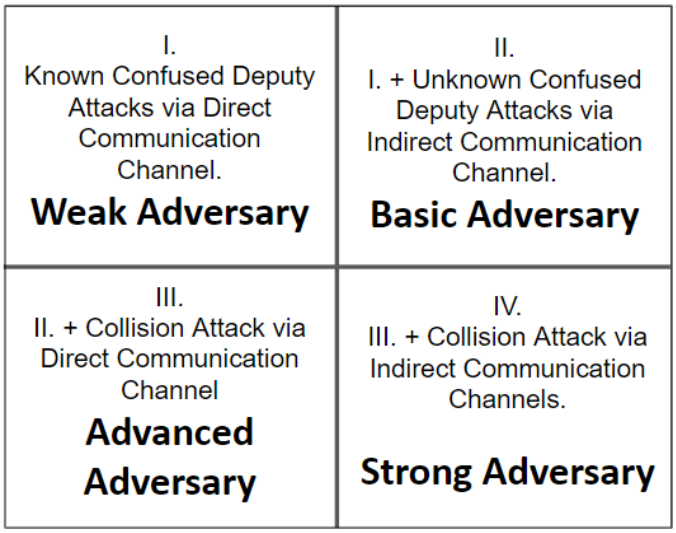

Fig. 4: XManDroid* - The threat model

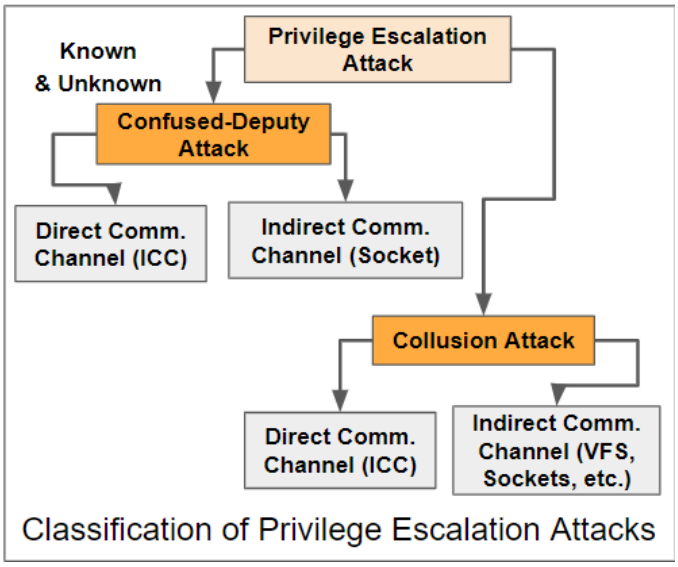

Fig. 5: XManDroid* - Categories of attacks and their communication channel

& collusion attacks, to understand the modeling challenges and design the security framework. Relying on the third-party applications for correct policy enforcement is not a viable option as they might themselves be malicious in nature. Thus, Bugiel et al. in [3], propose that there is a need to enforce security policies and access-control at the system level to protect against this kind of attack.

A. Problem Statement

The main contributions in this study are mentioned as follows:

- To investigate the design challenges faced by security community to protect against different variants of privilege-escalation vulnerabilities and define a standard threat model for the same.
- To propose a system-centric generic security framework to defend against such attacks by extending access-control mechanism at middleware and kernel level.
- To enable system-level security policy definition and enforcement for the proposed security framework and evaluate the effectiveness, performance and usability of the proposed framework.

B. Methodology

Based on the prior research works to defend against this attack (as discussed in section. II), the authors propose a generic threat model with a varying scope of the defense. As illustrated in Fig. 4 & 5, they categorize the attack into four threat models based on the following parameters: the class of attack, the channel chosen by the malicious application to delegate the task, and whether the application is already known to be susceptible the vulnerability. The ideal solution is expected to defend against the *Strong Adversary* by preventing all known and unknown attacks over both direct and indirect communication channels. However, it might lead to a high rate of false positives where the framework might deny the execution of even benign operations. Thus, a flexible and configurable threat model with varying level of protection provides a trade-off between the security and the usability of the proposed framework.

For this study, we consider all default and third-party applications as susceptible to confused-deputy vulnerabilities such that any malicious application may target them to execute security-critical operations. The underlying Android middleware and the Linux kernel is considered as trusted. Further, the design requirements for this framework are: *a*) a system-level defensive security mechanism to fight against malicious applications *b*) compatibility with legacy applications, and *c*) low performance overhead.

The proposed security mechanism is extended from the author's previous research work, XManDroid, in [20]. Fig. 6 illustrates the complete architecture of the extended XManDroid (or alternatively, XManDroid*). It consists of a system view which keeps track of the system state that includes the list of applications and the direct and indirect communication channels used by them. When a user installs any new application, the PackageManager updates the system view by adding the vertices and edges according to the communication pattern observed. The PolicyInstaller installs all the system policies and application-level baseline policies when the system (re)boots.

Whenever the *Reference Monitor* intercepts any ICC calls, it first verifies using underlying security mechanism, and when passed, it redirects the request to the *Decision Engine* for additional security check. The engine first checks for a suitable policy in the database, and if found, it appropriately enforces the decision on the request. In case the policy does not exist, the *Decision Engine* collects all the permissions assigned to the application, the settings configured as well as the current *System View* to determine whether or not the request is suspicious. At this point, the *Decision Engine* acts as the Permission Decision Point (PDP) and asks then *Reference Monitor* to enforce its decision. A simple example of a *System*

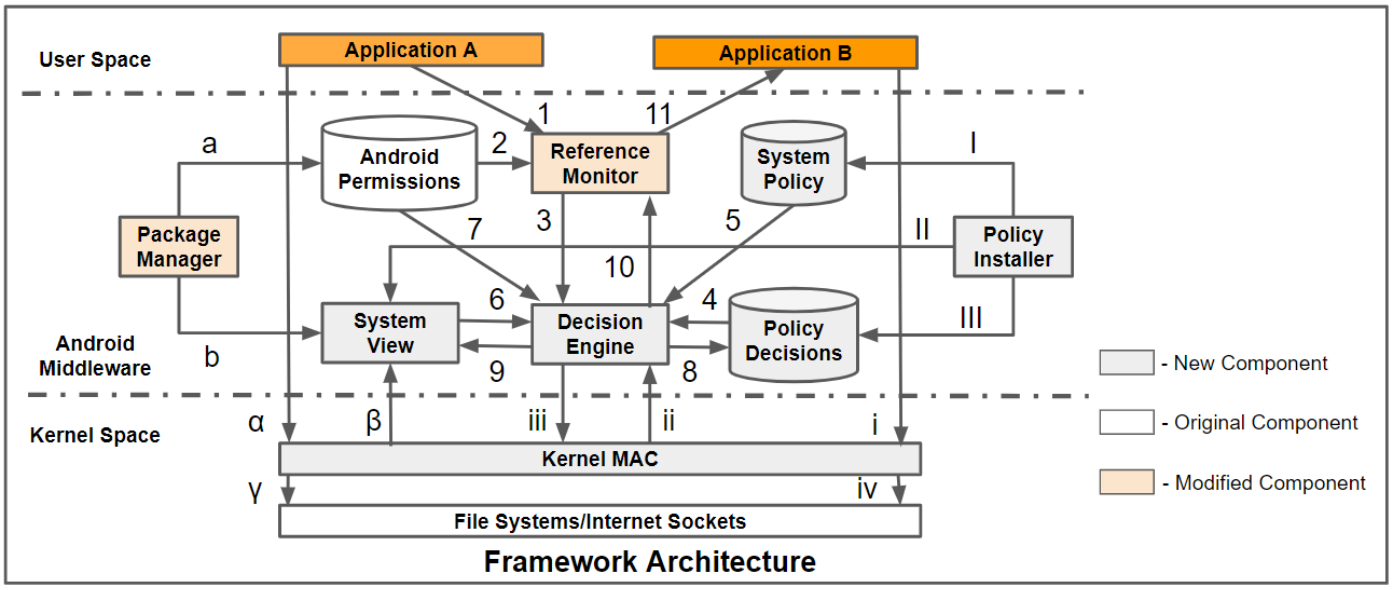

Fig. 6: XManDroid* Architecture - System-centric Security Framework

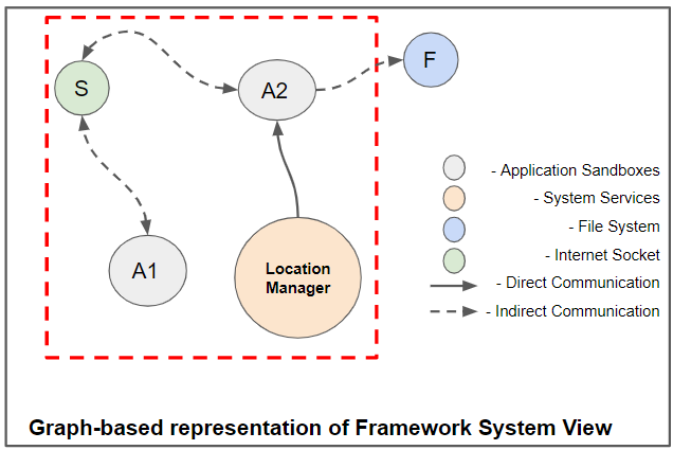

Fig. 7: XManDroid* System View - Graph representation of suspicious communication between applications

View Graph and potentially malicious communication channel between applications is depicted in Fig. 7. Here, A1 carries the permission to connect to the Internet but cannot access device location data while A2 can access location data as well as connect to the Internet. Thus, if A2 is vulnerable to privilege-escalation attack, A1 might be able to access location data from A2 and use it further for unintended purposes.

For communications through indirect channels, the *Kernel-MAC* intercepts the calls from the application and checks for a matching policy to carry out the required action. If such a rule exists, it allows the operation, or else, it queries the *Decision Engine* for the policy decision for the request. It further denies or grants the permission and updates the *System View* as well as internal policy database, whenever necessary. Additionally, to avoid confused-deputy attack arising from *Intents*, the authors also implement a system-centric call-chain mechanism, called *Intent tagging*, used to navigate to the current ICC in the *System View*.

To implement the proposed security framework, the authors add a few new components on top of the existing mechanism, as well as the existing components, are modified both at the kernel as well middleware level. *System View* is built using a graph library that initializes on the first boot of the device. The `PackageManager` and `ActivityManager` aids to the process by providing with the list of installed applications, the vertices and edges that correspond to their respective communication channels. The *PackageManager* is modified such that it updates the System View on the (un)installation of an application. For each tuple of {*caller_id, callee_id*}, the *Policy Decisions* are stored in the database managed by the Policy Installer & the Decision Engine. The *Policy Installer* installs or updates the *System Policies* as well as baseline Policy Decisions. The existing *Reference Monitor* is modified such that after verifying the communication request, it redirects the request to the Decision Engine for further verification. The *Decision Engine* then collects information from the System View, the existing policies and granted permissions and then determine whether or no the communication should be allowed. *TOMOYO Linux*, a path-based MAC implementation, is used to implement the kernel-level MAC which also allows a feedback channel between the kernel and the middleware for context-switching and an interface to fetch policies and update current system view. The system policies that are defined initially are based on the target threat model. The authors propose different policy profiles for varying level of protection, as desired for the user. Basic, Advance & Strong policies might affect the usability of the application and is suitable for higher level of security while default profile could be enabled in the system for all the users. These policies are defined using a formal security assurance language, called VALID.

C. Evaluation & Result

The authors manually evaluate the proposed security framework by including 25 test users and 50 target applications cho-

sen from different categories on the Play Store. The selected test users were instructed to (un)install applications, trigger as much functionality in any application, as possible, and thoroughly use them for better coverage of target operations. They then evaluate the access requests from applications in each of these devices and presents the result of the framework as follows.

Communication pattern among third-party applications: The authors conduct a heuristic analysis of the communication pattern from the logs obtained from the target devices. They observe that none of the target applications communicates among themselves or shares data over indirect communication channels such as filesystem or Internet sockets. It implies that the framework would be useful to identify and determine the intent of the request and its originating application when it tries to communicate using these channels. Further, the statistics also suggest that applications do not tend to share functionalities among themselves while they only share data using content providers. Thus, the direct communication patterns are also easily distinguishable for the proposed framework to understand and decide on the actual intent of the request.

Effectiveness: The authors evaluate the effectiveness of the framework based on the total false positive while denying communication and false-negatives such that whether or not it allows any suspicious communication to proceed. To test the framework and obtain the false-allowed and falsely-denied communication rate by the framework, they develop sample applications with suspicious requests, along with appropriate generic as well as specific rules required by the framework. While the framework results in no false-negatives, it does report a false-positives when the policies are too fine-grained. Thus, we observe that specifying generic configurations is best suited for this framework to defend against this vulnerability.

Performance & Usability: The authors analyze the performance of the framework and report that it imposes negligible runtime overhead on the runtime. While requests to *System Content Providers* causes a significant overhead of around 48% because of the multiple access checks that take place for each request, communication with other components such as System services or *Intents* do not pose any significant overhead for the user to notice. However, the framework might affect the usability of the applications when any benign communication is falsely-denied by the framework as it would hinder the expected functionality. Further, there would be no fail-safe defaults specified by the application developers as the permissions required to execute such operations is already expected to be granted by the user.

VI. Discussion

We observe that since both the lead studies aim to solve respective challenges in the older version of the Android architecture, it gives a historic, yet relevant, view of the problem statements. However, we emphasize on the fact that their work and propositions have found a place in the current Android security mechanism. From the privacy perspective, ever since Android v6.0, the user now has an option to selectively grant or revoke the permission of an application at runtime. Moreover, the users need not provide all the permissions upfront at the install-time and can decide upon the context of its usage. From Android v11.0, the user also has the provision to grant one-time permissions and can allow specific data access only when the application is in the foreground. It implies that indeed there is a need for flexible access-control mechanism at runtime that could be controlled by the user.

The authors in [2] suggests a few other key factors important to prevent privacy leaks. The permission framework could be improved to educate the user about the context(s) in which the application could use the permission as well as the privacy and security risks involved with it. Further, the involvement of end-user is a necessary parameter to adequately improve the permission framework and preserve the contextual integrity of applications. The authors suggest that a generic privacy policy template could be defined with the help of user studies and heuristic analysis to enforce default privacy-preserving policies on the untrusted applications.

Following the suggestion discussed above, Raval et al. [12] proposed an extensible permission plugin for Android applications, DALF, that enables the end-users to control the data access from different applications. However, malicious applications may bypass the plugin check if they can exploit the privilege escalation vulnerability exposed by other installed applications. Further emphasizing on the importance of contextual integrity and user-involvement, Wijesekara et al., in [9], added that the visibility of the access requests as well as their frequency are pivotal for the user to decide whether or not to allow certain operations to execute.

Chitkara et al. & Diamantaris et al. target similar problem statement in their recent research work in [13] & [14], respectively. They argue that the fact that, nowadays, applications contain numerous third-party libraries for different purposes, the privacy risks associated with the application as well as the user data it handles has increased more than ever. In the former work, the authors propose an Android application, *ProtectMyPrivacy*, which aims to intercept data access requests and determine the origin of the request, i.e., whether the request actually originated from the application or from the third-party libraries and further aids the user to decide on the request by providing additional contextual cues. They point out that approximately 30 third-party libraries are used more than 50% applications and leak private data. While, in the latter work, the authors propose a dynamic analysis framework, called *Reaper*, that identifies the origin of the request, as above, at runtime. The proposed model could be used by both the end-user as well as part of the official vetting process at the Play Store. They also mention that almost 65% of the permission requests originate from the libraries within the application, of which around 37% of them uses it for tracking, advertisement and user profiling purposes, obviously undesired by the user.

We also discussed how the access rights given to benign or malicious applications could also be abused by other applications to carry out security-critical operations without even leaking any personal data, and thus, may lead to unwanted ramifications. The authors, in [3], propose a standard and generic threat model that captures both confused-deputy as well as collusion threats exploited over different communication channels and argues for a system-centric security mechanism to prevent any application-level intervention in security policy enforcement. We observe that though, the security framework put forth yields positive results and block suspi-

cious communication instantly at runtime, there is a significant trade-off between usability and security while monitoring at runtime, from users' perspective as the proposed models, so far, does incur some performance overhead as well as leads to false-positives, depending upon the policy configured for the framework. On the other hand, static vetting mechanisms fail to model the user requirements and often over-approximates the benign functionalities as suspicious.

Following their work above, Bugiel et al. proposed an extended security architecture, called *FlaskDroid*, that aimed to provide policy-driven fine-grained mandatory access-control at both middleware and kernel level. These system-defined policies are inspired and extended from SELinux *type enforcement* rules and can provide privacy as well as security guarantees at the middleware level. However, it does not guarantee to block malicious collusion of applications over indirect channels. Further, Backes et al. introduced SCIPPA that aims to prevent against confused-deputy attack over ICC by extending the Binder-IPC with provenance information. This provenance information could be used to identify the actual origin of the request and thus, enforce appropriate actions. The limitations of this approach are apparent that it could only cover a specific variant of the security threat in discussion and also does not provide any contextual cues to the user in case of private data access.

Further, Elish et al., in [17], studied the feasibility and performance of automated policy-driven approaches to detect maliciously colluding applications over benign applications. They conduct their study over 2.6K applications by building pairwise maps between ICC channels of these applications and report that over 84% of them communicate to other built-in or third-party applications over intent-based channels, contrasting to the findings of the lead study. We assume that there is significant increase in the figures because of increased functionality, task delegation and third-party libraries among applications. They further report that this approach yield high false-positives and labels benign communication requests as malicious. The results obtained establishes the observation in our lead work that these security policies might intervene the normal execution of benign applications and falsely label them as suspicious.

Seo et al. proposed a security framework, called FlexDroid, in [18]. The fact that the third-party libraries within the untrusted application also enjoys the same level of access rights as the parent application is a potential threat to the privacy of users' personal information as well as security of the application. The authors in this work try to solve this design issue by monitoring the operations at runtime. More specifically, the proposed solution inspects the stack and cover the dynamic code execution at runtime, thus segregating between the process calls originating from the core application and the ones that originate from the 3rd-party libraries embedded within. The framework then provides the user with the contextual information to allow or deny the access request accordingly.

From all the research works discussed above, we understand that the complexity of the application has increased with time and it imposes new challenges to design a full-proof solution and defend against privacy and security threats to the user. Though there exists no framework that completely solves the issue, a combination of these approaches might help to eliminate the threat at some cost of usability of the application.

VII. Conclusion

In this seminar report, we discussed the privacy and security threats that untrusted third-party applications might pose to the end-user. We elaborated on two specific research contributions to prevent privacy leaks and thwart against privilege escalation vulnerabilities in the wild, respectively. We then also reviewed later research works closely associated with our lead contributions to determine the prevalence of such threats in the wild in the current scenario since the lead study dates back to 2011-12.

We observe that untrusted applications often exploit the permissions granted to then by leaking private information such as location, IMEI number, contact information, SMS, from the users' device out of the benign context. We also observe that communication patterns for inter-process exchange are quite distinctive between the malicious and benign context of usage. We understand that with increased third-party components such as plugins & libraries as well as the complexity of the architecture, there is a need to further segregate the privileges and identify the origin of the operation to decide its fate. Thus, it is practically difficult to design and implement a full-proof solution to such real-world issues and also without hindering the user experience as well as the normal execution of benign applications. The key takeaway from this work is that while no existing framework is entirely robust and efficient against privacy leaks and privilege escalations. However, by involving user feedback, a standard access template could be agreed upon based on the context of the usage of the permission to prevent privacy leaks. Similarly, system-level privilege segregation for application components and policy enforcement on inter-process communications could yield favorable results.

References

[1] Statista. Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 2nd quarter 2020. [Online]. Available: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/

[2] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *International conference on Trust and trustworthy computing*. Springer, 2011, pp. 93–107.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android." in *NDSS*, vol. 17, 2012, p. 19.

[4] 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)*, vol. 32, no. 2, pp. 1–29, 2014.

[5] W. Enck, M. Ongtang, and P. McDaniel, "Mitigating android software misuse before it happens," 2008.

[6] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," *Security and Communication Networks*, vol. 5, no. 6, pp. 658–673, 2012.

[7] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses."

[8] M. Nauman, S. Khan, and X. Zhang, "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*, 2010, pp. 328–332.

[9] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 499–514.

[10] D. Votipka, S. M. Rabin, K. Micinski, T. Gilray, M. L. Mazurek, and J. S. Foster, "User comfort with android background resource accesses in different contexts," in *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, 2018, pp. 235–250.

[11] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. A. Zhang, N. Sadeh, Y. Agarwal, and A. Acquisti, "Follow my recommendations: A personalized privacy assistant for mobile app permissions," in *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, 2016, pp. 27–41.

[12] N. Raval, A. Razeen, A. Machanavajjhala, L. P. Cox, and A. Warfield, "Permissions plugins as android apps," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019, pp. 180–192.

[13] S. Chitkara, N. Gothoskar, S. Harish, J. I. Hong, and Y. Agarwal, "Does this app really need my location? context-aware privacy management for smartphones," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, pp. 1–22, 2017.

[14] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "Reaper: Real-time app analysis for augmenting the android permission system," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 37–48. [Online]. Available: https://doi.org/10.1145/3292006.3300027

[15] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 131–146.

[16] M. Backes, S. Bugiel, and S. Gerling, "Scippa: system-centric ipc provenance on android," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 36–45.

[17] 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.

[18] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "Flexdroid: Enforcing in-app privilege separation in android." in *NDSS*, 2016.

[19] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones." in *NDSS*, vol. 11, 2011, pp. 17–33.

[20] 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.