Invetter: Locating Insecure Input Validations in Android Services

Lei Zhang
Fudan University
lei_zhang14@fudan.edu.cn

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

Yuyu He
Fudan University
heyy16@fudan.edu.cn

Zhenyu Zhang
Fudan University
zhenyuzhang15@fudan.edu.cn

Zhiyun Qian
University of California Riverside
zhiyunq@cs.ucr.edu

Geng Hong
Fudan University
ghong17@fudan.edu.cn

Yuan Zhang
Fudan University
yuanxzhang@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

ABSTRACT

Android integrates an increasing number of features into system services to manage sensitive resources, such as location, medical and social network information. To prevent untrusted apps from abusing the services, Android implements a comprehensive set of access controls to ensure proper usage of sensitive resources. Unlike explicit permission-based access controls that are discussed extensively in the past, our paper focuses on the widespread yet undocumented input validation problem.

As we show in the paper, there are in fact more input validations acting as security checks than permission checks, rendering them a critical foundation for Android framework. Unfortunately, these validations are unstructured, ill-defined, and fragmented, making it challenging to analyze. To this end, we design and implement a tool, called `Invetter`, that combines machine learning and static analysis to locate sensitive input validations that are problematic in system services. By applying Invetter to 4 different AOSP codebases and 4 vendor-customized images, we locate 103 candidate insecure validations. Among the true positives, we are able to confirm that at least 20 of them are truly exploitable vulnerabilities by constructing various attacks such as privilege escalation and private information leakage.

KEYWORDS

Android Framework, System Service, Input Validation, Permission Validation

1 INTRODUCTION

Never before has any operating system (OS) been so popular as Android. Over 60 percent [22] of mobile devices are running Android with a huge number of applications (apps for short) that are connected to our daily life. To achieve a variety of functionalities, apps read and manipulate Android system resources, such as GPS device and screen display, and perform sensitive operations such as sending and erasing SMS messages. In Android, these resources and sensitive operations are administered by over 100 system services. Evidently the access control in these services plays an important role in the security of Android systems and is a relatively understudied research area.

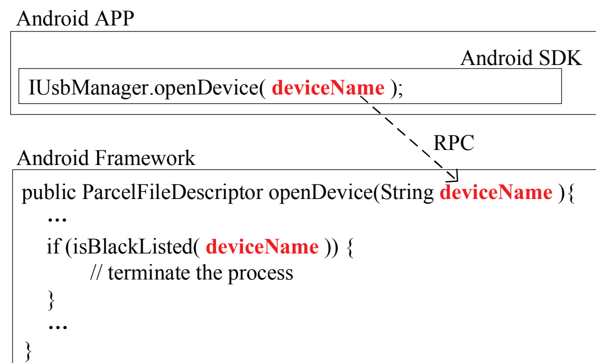


Figure 1: Input validations from Android system service and Linux kernel.

Among the access controls, permission validations are well-studied, e.g., Kratos [26] addressed the inconsistent permission enforcement problem. In this paper, we conduct an empirical study on a different set of critical security enforcements in system services, which we define as *sensitive input validations.* As will be unveiled in our study, Android imposes over 700 distinct sensitive input validations (only a lower bound), compared to just 351 permissions. They serve various purposes, as an example in Figure 1, the sensitive input `deviceName` is used to restrict usage of sensitive operations, thus preventing system services from being abused by untrusted apps. To the best of our knowledge, our work is the first to systematically study the secure use of sensitive input validations of Android services.

Unlike the traditional input validation studies that focus on a narrow and well-defined set of sensitive input, e.g., web input that can cause SQL injection attacks, and user-space pointers passed to Linux kernel that can cause memory corruption attacks, our paper focuses on the opposite end of the spectrum where it is not even

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243843

clear what input crossing the trust boundary should be considered sensitive and therefore checked:

- **Unstructured.** Unlike Android permission checks that rely on system-defined interfaces, e.g. *Context.checkCallingOrSelfPermission()* and *Binder.getCallingUid()*, sensitive input validations in system services are difficult to identify. In fact, as illustrated in Figure 1, any input parameter to a public method of a service can potentially lead to a sensitive input validation (a conditional statement involving the check of a parameter).
- **Ill-defined.** Unlike permission validations, which are well documented by the Android permission model [13], no publicly available sources define how sensitive input validations should be carried out in Android services. Thus, it is unclear whether an input needs to go through validation and whether it is done correctly.
- **Fragmented.** Sensitive input validations are dispersed in a large number of Java classes. For example in Android 7.0 , our evaluation shows that they are scattered widely in 173 different Java classes, while Android permission enforcements are clustered in 6 classes. Moreover, even in the same service method, sensitive input validations are commonly scattered in various execution paths, restricting system operations in a fine-grained manner.

Despite the importance of sensitive input validations in Android services, their design and usage have not been well thought out, evidenced by its ad-hoc nature outlined above. By attempting to summarize and identify flaws related to sensitive input validations, we make two observations as below (which are detailed in §3).

- **Confusions about system security model.** Android services sometimes incorrectly trust data from apps without any validation. Interestingly, we even find sensitive input validations sometimes misplaced in the Android SDK (which runs as the same process of the app), demonstrating a complete misunderstanding of the trust model.
- **Weakened validations in customized system images.** In the Android ecosystem, system services are often customized to provide added value. During the process of customization, we find common problems where the sensitive input validations may become weakened.

By designing a general machine learning technique to identify sensitive input validations as well as using static analysis to identify their problematic uses, we develop `Invetter` and evaluate it on both Android AOSP system images and third-party customized images. According to our analysis of 4 AOSP images and 4 third-party customized images, we find at least 20 exploitable vulnerabilities. For example, we show that a zero-permission app (running in the background) can stealthily launch phishing attacks, steal a user password stored in another app, and sometimes delete the entire "system" directory. Many of these cases are demonstrated in our anonymous video: https://youtu.be/erLY_OMi4kQ.

Contributions. The contributions of our work are summarized as follows.

- Our work is the first to systematically analyze, identify, and report the scale of sensitive input validations inside Android system services and their potential flaws.
- From analyzing and summarizing the flaws of sensitive input validations, we develop a fully-functional tool `Invetter` to automatically discover their problematic uses, which we plan to open source.
- We evaluate our tool on 4 AOSP images, from Android 5.0 to 8.0, and find 20 exploitable vulnerabilities in total, many of which are confirmed by the corresponding vendors.

2 BACKGROUND

In this section, we provide necessary background for understanding how the Android system services work, and how input validations are performed in Android framework.

2.1 Android System Services

The Android framework consists of more than one hundred system services which provide support for accessing various system resources, such as retrieving user location, sending SMS, and checking network connectivity. Since these services are part of the Android framework, their execution environment enjoys more privilege and are separated from apps. For example, the system service media is executed in a system process called *media_server*. Commonly, system services should be registered to the *ServiceManager*, so that they can be accessed by apps or other services.

Each system service can be accessed via a set of pre-defined public interfaces. These interfaces are commonly declared using Android Interface Definition Language (AIDL). During the compilation process of Android framework, interfaces declared by AIDL are compiled into two sets of Java classes, the *Stubs* and the *Proxies*, to act as a channel between services and their clients (which can be apps or other services). Specifically, *Stubs* are extended by the services to implement their functionalities, and *Proxies* encapsulate the remote-process communication (RPC) logic to facilitate easy access by the clients.

Figure 2 depicts this process. To initiate a request to a service, the client must first send a query to the Android *ServiceManager*, which maintains a mapping between services and their corresponding Binder objects. Using the Binder object returned by the *ServiceManager*, requests can be served using the interfaces defined by the *Proxies*. *ServiceManager* has no way of forbidding apps from forging their inputs, thus in principle it should not trust any apps-supplied data.

Additionally, on top of the *Proxies* abstraction, Android SDK provides a set of *Managers* as wrappers that provide another layer of APIs which are even simpler for developers to use. Different from the service code, *Managers* execute in the same process as the running app, so malicious developers can reimplement and overwrite them. Thus, system services cannot trust any security validation in such app-controlled code.

2.2 Sensitive Input Validations in Android Services

Sensitive input validation acts as a critical part on the security of Android services. Commonly, input validation looks like the

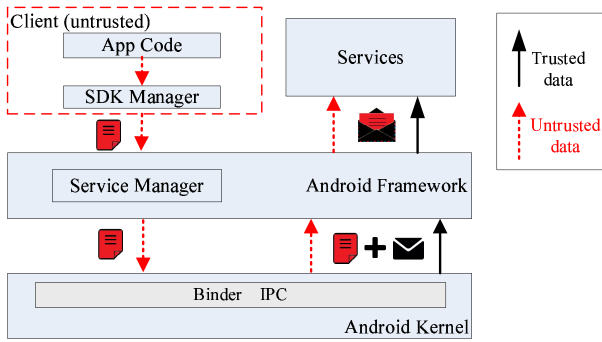


Figure 2: The Binder-based RPC between system service and its client.

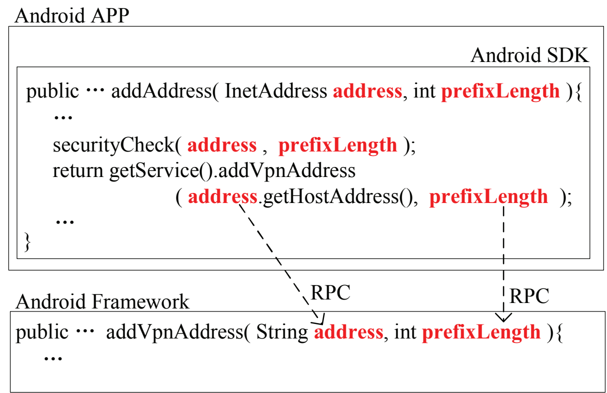
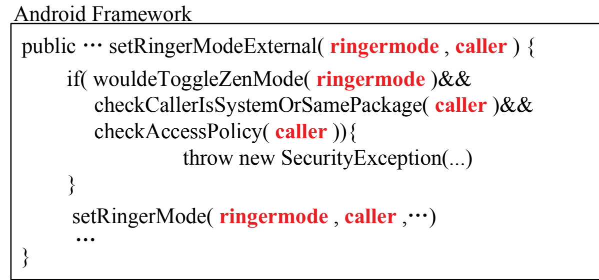
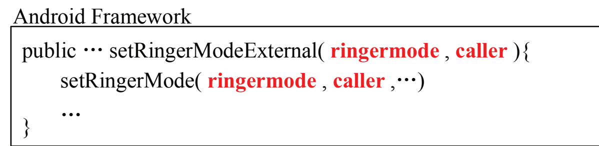


Figure 3: An input validation in Android SDK.



(a) Code in AOSP



(b) Code customized by Xiaomi

Figure 4: An input validation in Android Audio Service is removed in customized image.

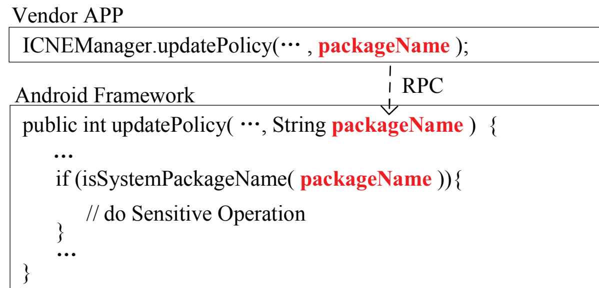


Figure 5: An input validation in system service which trusts the inputs from less privileged apps.

following: a piece of input data is compared against a set of predefined expectations, or cross-validated with trusted data source, and some subsequent actions will be taken based on the outcome of the comparison. Note that not all input validations are for security purposes, e.g., checking the format of input or whether there is a null pointer. In this paper, we are more interested in security-focused validations.

In Android, we summarize them in two forms: (1) verify the identity/property of input sender, or (2) restrict the usage of sensitive resources. For (1), typically the identities/properties can be either well-known: *uid, pid, package name*, or obscure: *token, cert*, and so on. For (2), an example is the URIs used as keys to access system content providers which can be restricted by checking the scope of the URI supplied by an app.

3 OBSERVATION: INSECURE INPUT VALIDATIONS

By analyzing the existing sensitive input validations, we observed two sources of insecure input validations:

Confusions About System Security Model. As described in §2, system services enjoy more privilege, e.g. a system uid compared to apps and should not blindly trust any data sent from an app.

However, we observed that many system services not only trust app-supplied data from *Managers* (wrappers provided by SDK), but also misplace sensitive input validations in the *Managers* code. For example, Figure 3 illustrates a mistaken trust of app-supplied data. Since apps can bypass the *Managers* and forge their inputs to system services (*address and prefixLength* in this example), the security check does not operate as expected. This allows any app to insert new VPN server addresses into the system, which can potentially redirect all of the device's traffic to an attacker without authorization.

Weakened Validations In Customized System Images. In the Android ecosystem, system services are often customized to provide added value. During the process of customization, the input validations may become weakened. Within the 4 customized images we studied, 35 system services are modified, with 41 input validations affected. Figure 4 depicts an example of weakened sensitive input validation. Since the customized image of Xiaomi removed the check before invoking *setRingerMode()*, any app is free to change the ringer mode arbitrarily (e.g., silent or vibrate).

On the other side, over 203 new customized services are introduced in these images, together with 326 new sensitive input validations. We find even more suable and interesting confusion cases. For instance, shown in Figure 5, Qualcomm introduces a new service called *CNEService*. Although its privileged interfaces are designed to be available to only its vendor-supplied system apps, it cannot be guaranteed at all (i.e., the packageName parameter is completely untrusted).

4 METHODOLOGY

This section introduces our methodology to discover insecure input validations in Android. We will give an insight that guides our system design, the overall picture of our system, followed by each component in detail.

4.1 Insights and Workflow

In an ideal world where we have the labels of all the sensitive inputs (parameters of public service methods), all we need to do is to identify the absence of validations against those inputs. In practice, unfortunately, such labels are never provided by developers and at best have to be inferred which is generally an open problem. Therefore, we take a different approach — instead of relying on identifying all sensitive inputs and their missing validations, we can look for existing sensitive input validations that are misplaced or incomplete, which is a much more tractable problem. The assumption is that the probability that a sensitive input is never validated anywhere in the entire Android codebase is small, and hence we argue that locating existing sensitive input validations and their insecure uses can still capture a significant fraction of the related vulnerabilities. We admit that this assumption is difficult to validate as the ground truth of the total number of vulnerabilities is hard to obtain.

Invetter operates in three steps, as illustrated in Figure 6. First, Invetter thoroughly extracts system services along with their public interfaces from a given Android image, and recognizes all input validations using a structural analysis. Second, these extracted validations are passed into our learning module to recognize a subset of them that are "sensitive input" validations. It is worth noting that even though locating existing sensitive input validations is a simpler problem than identifying all sensitive inputs in the world, the very problem is still challenging. This is because (as discussed in § 1), sensitive input validations in Android are unstructured, ill-defined, and fragmented, and no simple structural patterns can capture them. Finally, we look for insecure input validations based on our observations introduced in §3. These reported cases are then considered as candidate vulnerabilities, which will be further verified by security analysts.

4.2 Extracting Input Validation Structures

Since input validation is the centerpiece of our analysis, we need to automatically identify and study input validations in Android framework, which is a challenging problem; this is because they are neither performed through pre-defined system interfaces, nor identifiable via fixed APIs like permission checks.

We leverage the inherent structural characteristics in input validations. Specifically, different from general branching statements, an input validation not only compares the input with other data, but also terminates its normal execution immediately when the validation fails. For example, a *SecurityException* can be thrown as a termination action. Figure 7 illustrates two input checks from the Android framework, in which one (a) is an input validation and the other (b) is a normal branching. In Figure 7.(a), the system service verifies the uid of the calling app, and throws an exception to stop the execution of the system methods when the validation fails. In comparison, Figure 7.(b) only aims to handle different kinds of input and select the corresponding handler method.

Based on this observation, we need to understand which set of termination actions are typically taken if a validation fails. To reiterate, the first requirement of input validation is that the input must be propagated to a comparison statement through data flow and compared against some pre-configured values or results dynamically retrieved from other APIs. Then, different actions are taken based on the comparison result. After analyzing a handful of real-world input validations in Android, we summarize the following four kinds of termination actions:

- **Throw exception.** A straightforward way to show that the client fails in the input validation is throwing a specific exception, such as *SecurityException* and *IllegalArgumentException.*
- **Return constant.** System services use some pre-defined constants to indicate that caller fails in input validation, which will be returned in the termination actions.
- **Log and return.** Logging information is useful in monitoring the running of the system. In termination actions, they commonly log some information about the illegal input and then return.
- **Recycle and return.** In some cases, before the exit of execution, public interfaces need to recycle the previous allocated resources.

In some cases, some input validations are simply data format checks, e.g., a *Null* object check. Since this kind of validation does not lead to serious security consequences (other than perhaps crashing the system service if missing), we choose to exclude this kind of validations in our framework and focus on other non-DoS-related vulnerabilities, e.g., privilege escalation or privacy breach.

By recognizing the termination actions, we can identify input validations with the following four steps: First, for a given system method, we obtain all conditional statements in the method body. Second, we identify the conditions that involve variables related to the method input (via data flow analysis). Third, we apply the filter to eliminate data format related validations. Finally, our analysis ensures that each recognized validation has a termination action. Our results described in §6.1 show that this approach can identify 800 input validations in Android services with only 71 false positives.

4.3 Learning Sensitive Input Validations

Unfortunately, no structural patterns can tell sensitive input validations from other less sensitive ones. A precise and complete analysis would require inferring the semantic significance of the input variables in terms of how they are processed in the service and

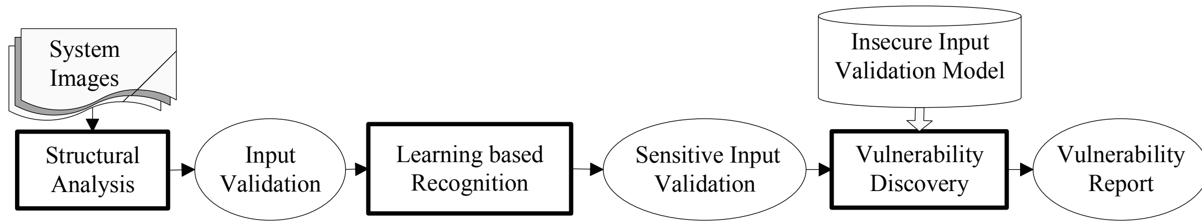


Figure 6: The overall architecture of Invetter.

```
boolean securityViolation =  ai.uid != 0
              && ai.uid != Process.SYSTEM_UID
 if (securityViolation) {
        String msg = "Requesting code from " + ai.packageName
                 + " (with uid " + ai.uid + ")";
        throw new SecurityException(msg);
}
```

(a) Input Validation

```
String action = intent.getAction();
 if (ACTION_PASSWORD_CHANGED.equals(action)) {
        onPasswordChanged(context, intent);
} else if (ACTION_PASSWORD_FAILED.equals(action)) {
        onPasswordFailed(context, intent);
} else if (ACTION_PASSWORD_SUCCEEDED.equals(action)) {
        onPasswordSucceeded(context, intent);
} else if  ...
```

(b) Functionality check (not an input validation)

Figure 7: Code snippets of input checks within Android framework.

what kinds of operations they authorize. We consider this analysis to be infeasible as it requires a significant knowledge base describing what operations in the system are sensitive, which itself is difficult to obtain.

We take a drastically different approach through machine learning. The idea is to take advantage of the fact we can label a much smaller set of sensitive input validations as training samples, and have the machine learning automatically learn the rest.

We first present a strawman approach, which does not quite work. In Figure 5, we illustrate a simple example where a sensitive variable "packageName" is validated to check the identity of the caller package. One might imagine a natural language processing based technique to infer the meaning/sensitiveness of an English word. However, Android framework manages plenty of system resources, and uses a diverse set of variable names to represent different pieces. It is almost impossible to determine the sensitiveness of such domain-specific names without a complete understanding of Android framework.

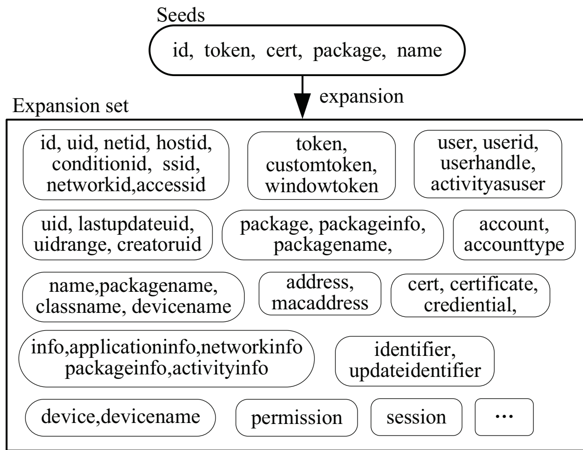
Instead, Invetter chooses to use the association rule mining technique [28] to automatically discover additional input validations that are likely also sensitive based on their co-occurrence with known sensitive input validations. The intuition here is sensitive input validations are often co-located in the same service methods. Taking the "packageName" and "uid" as an example, Android framework often uses them together to verify an app's identity (See Figure 9). They are thus likely to be positively correlated in terms of their sensitiveness. Our detailed approach is introduced below.

4.3.1 Grouping input validations for association rule mining. One important requirement in association rule mining is that we need to observe enough samples/occurrences of any given variable. However, if we treat each unique variable name separately, we may end up with cases such as variables `flag1` and `flag2` which each appear only one time respectively in the code base, disallowing effective association rule mining. Our intuition is that if the variables share a common term (or prefix/suffix), they must be semantically related and we can simply group them together. To do so, we go through a series of steps:

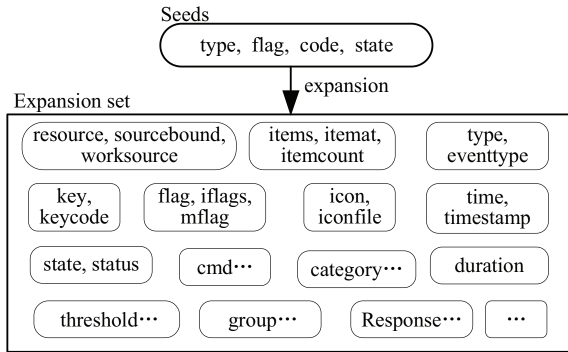
Word splitting and stemming. Normally input parameters are letter-case separated words. For example, 'componentName' can be separated as 'component' and 'name', and 'groupOwnerAddress' can be separated as 'group', 'owner', and 'address'. Based on this approach, we split such long words into separated words. Furthermore, for each separated word, Invetter attempts to further identify a single common root or base word. For example, words like 'types' and 'subtype' stemmed from the base word 'type', and the prefix 'm' of words 'mflag' and 'mname' should be removed also. To find the base word, Invetter splits words by iteratively matching the maximum length word in WordNet [19] until the input word cannot be further split, and discards the remaining. After this step, Invetter obtains the root words of each input parameter.

Variable name normalization. We can obtain a normalized name by merging the root words of each input parameter. However, even though word splitting and stemming are applied, meaningless qualifiers are unavoidable, skewing the final name. For example, variable 'linkaddress' is split into 'link' and 'address', while both 'address' and the qualifier 'link' are treated as root words. To remove the qualifiers, Invetter calculates the occurrence frequency of each pair of words. If two words often occur simultaneously, we only retain the more popular word. After this step, we can group variables based on their normalized names, which will facilitate the association rule mining.

4.3.2 Learning new sensitive input validations. In total, we obtained over 1132 input validation groups after the above step. However, without a priori knowledge, it is not clear whether a validation involves any sensitive input. Fortunately, we observed that developers tend to enforce similar input validations in adjacent places. For example, in Figure 9, various sensitive input validations



(a) Verify caller identity



(b) Restrict sensitive resource usage

Figure 8: The Initial seeds and expanded groups for recognizing sensitive input validations.

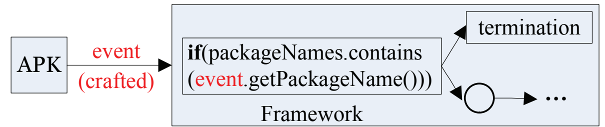
are enforced nearby. Thus, we can figure out a small number of sensitive input validation groups, and discover other related groups.

Seeds of sensitive input validations. As described in §2.2, only the input validations which verify the user identity, or restrict the usage of sensitive resources, are considered sensitive. Thus, we curated the list of input validation groups in Figure 8 as the initial seeds.

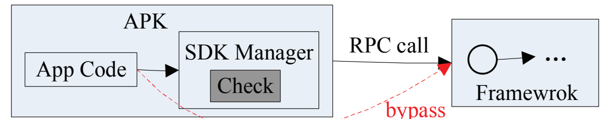
Association rule mining. We expand the sensitive input validation sets by conducting the association rule mining. First, we calculate the distance between each pair of input validations. Specifically, if two input validations occur on two basic blocks with a common edge, we consider these input validations adjacent to each other. Then, if two input validation groups contain three adjacent pairs, we associate these groups together (number chosen empirically). Finally, starting with the seeds, we collect all the associated groups iteratively until no more new group can be discovered. Figure 8 shows the partial list after expansion. As we can see, the technique is effective in discovering a large number of groups of sensitive input validations.

```
...
if ( uid != Binder.getCallingUid ){
        throw new SecurityException( "··· " ) ;
}
if ( !isSystemPackageName(packageName) ){
        throw new SecurityException("··· " ) ;
}
...
```

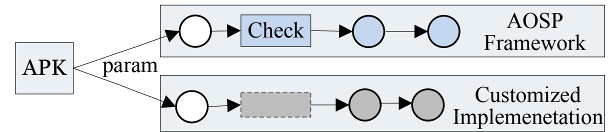
Figure 9: An example for nearby input validations.



(a) Incorrectly trusting app-supplied data



(b) Incorrectly trusting code in the app process



(c) Weakened sensitive input validation in customized images

Figure 10: Types of security flaws in sensitive input validation.

4.4 Vulnerability Discovery

Invetter operates from two independent perspectives: by searching for incorrect/insecure sensitive input validations in each Android system image; and by comparing inconsistent security enforcement between different images. In this section, we first describe our intra-image analysis followed by the inter-image analysis.

4.4.1 Intra-image analysis. Based on our observation in §3, we reason about other possible incorrect assumptions that affect sensitive input validation. We summarize them as illustrated in Figure 10.(a) and (b).

Incorrectly trusting app-supplied data. Some services validate the caller identity based on input parameters that can be easily manipulated by untrusted apps. Clearly, the input parameters can be originated from untrusted apps and cannot be trusted for sensitive input validation. Based on the expanded input validations in Figure 8, an sensitive input validation is considered vulnerable if it verifies an app-supplied sensitive data, and our learning based sensitive input validation analysis reveals that it is applied to check the identity of the caller.

Incorrectly trusting code in the app process. Unlike permission checks which never occur in the application process itself, input validations are actually quite often misplaced due to their unstructured nature. Specifically, we find that the collection of *Managers* in the Android SDK (see Figure 2) that run inside the application processes often acts as a proxy that packages data from the app and forwards them to Android service processes. During the data packaging process, these *Managers* also conduct input validations (many of which are sensitive).

We consider a case vulnerable when sensitive input validations are performed in the Android SDK and yet the Android services do not perform the same sensitive input validations (if both sides perform the same sensitive input validations then it is still secure). The scope of Android SDK includes not only the public interfaces, but also those that are labeled by *@hide* or *@SystemApi*, since apps can access these hidden interfaces with reflection.

4.4.2 Inter-image analysis. Given the set of input validations, we locate the corresponding public interfaces that deploy these validations. To locate the weakened sensitive input validations during vendor customization, we figure out inconsistent sensitive input validations between AOSP and customized images, as illustrated in Figure 10.(c). We first group the public interfaces of different system images, based on similarities in their method behaviors. Specifically, we borrow the techniques from [36] that represent a method behavior based on its data dependency graphs. We determine two methods to be similar when the similarity score is higher than 0.7. Specifically, the threshold is determined empirically from a small-scale experiment where we apply our inconsistent validation detection with AOSP as baseline to 4 third party images with three different thresholds (0.6, 0.7 and 0.8). As illustrated in Figure 11, 0.7 is the largest threshold that Invetter can find similar but not identical methods. Anything above (e.g., 0.8) is too strict and can find only unmodified methods. Then, by comparing the enforced validations inside each group, Invetter reports inconsistent access controls among different system images.

Note that it is insufficient to simply look at the class names and method names to determine similarity. The reason is that many vendor-customized services introduce new system services, which achieve similar functionalities as AOSP services yet with reduced security. We suspect that this is often times for reasons of convenience — it may be complex to change the existing AOSP services directly.

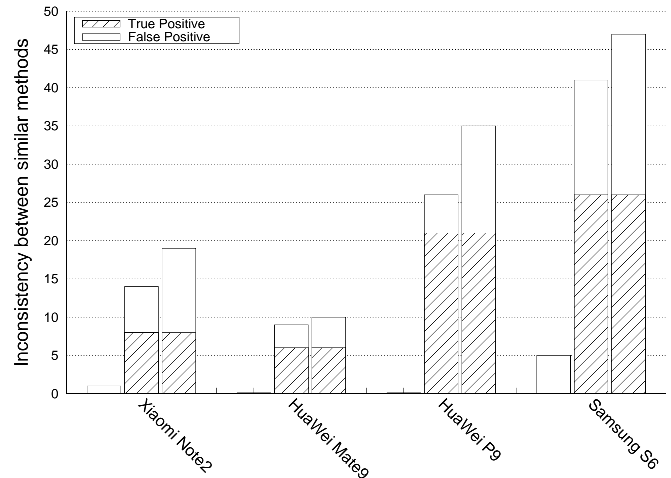


Figure 11: The results of evaluating different thresholds in inter-image analysis. The three columns present the results under different thresholds (0.8, 0.7, and 0.6 respectively).

```
IBinder iBinder = ServiceManager.getService("accessibility");
IAccessibilityManager service =
        IAccessibilityManager.Stub.asInterface(iBinder);
```

Figure 12: An example of client-side code for building a Binder-based RPC connection to a system service.

5 IMPLEMENTATION

We implement Invetter on an open source static analysis framework, Soot [24], with about 12,000 lines of Java code, which we plan to open source. Our implementation follows the same workflow as illustrated in Figure 6. In this section, we discuss a few major technical issues in implementing Invetter.

Extracting system services. The third-party system images are commonly not open-source. Thus, to analyze their code, Invetter uses Java bytecode as input. As the first step of our analysis, we extract Java classes from system images for analyzing. For a specific system image, Invetter first dumps all *dex*, *odex*, and *oat* files from it. Then, by using *oatTodex*, Invetter translates *oat* files and *odex* files into *dex* files. Next, the *dex* files can be processed by a tool *dexTojar*, which translates them into jar files, in which the corresponding Java class files are zipped. Invetter uses the Java bytecode extracted from the jar files as the analysis target.

A critical technical challenge for Invetter is to extract a complete list of system services. At runtime, all system services register themselves to Android *ServiceManager* so that the system can start all available services when initialized. However, there is no direct way to obtain the service list statically. Related work [26] searches the specific register interfaces, such as *addService* in *ServiceManager*. But in this way, only services registered in Java code can be extracted. Unfortunately, we find that many system services are registered to the system in native code. Additionally, smartphone vendors may customize their own service managers and register methods. Thus the system services in customized images cannot be completely identified with this approach.

We propose another approach based on the observation that if a system service is available to use, there should be some code in the system (e.g., Android SDK) that visits its public interfaces. Specifically, we identify a system service by finding one of its clients in the Java bytecode, that is, recognizing a client-side *Proxy* for Android Binder-based RPC. Figure 12 depicts an example of this. Besides, we also use the register interfaces as a supplement.

Extracting public interfaces. For each system service we find, we extract all its app-accessible public interfaces. Specifically, two kinds of public interfaces are considered in this paper. First, as aforementioned, methods declared by AIDL are public interfaces of

services. Thus, they are extracted as targets of our analysis. Second, public interfaces documented by Android SDK are also extracted. These interfaces are APIs (of various `Managers`) executed in app-controlled processes (see Figure 2). We utilize these interfaces to find misplaced validations described in §3.

Constructing control flow graph. Since Invetter conducts its analysis based on the control flow graph of Android framework, complete and precise call graph and control flow graph are essential for our approach. Invetter uses inter-procedure analysis to achieve better coverage and accuracy, thus it requires both intra-procedure information about how the execution flows inside the methods, and inter-procedure call information. To construct complete call graphs and control flow graphs, we first leverage the approach proposed in Axplorer [5], to connect the callers and their callees of asynchronous or implicit function calls. Besides, we utilize Spark, to generate points-to and class-hierarchy information, and to recognize possible referenced object types for each method call.

Conducting path-sensitive analysis. Path-sensitive analysis is often prohibitively expensive to apply in complex systems. Invetter requires inter-procedure analysis to cover inter-method execution paths, which further expands the search space. While there are many system methods that are relatively simple and can be handled, some methods have complicated control flow graph and generate plenty of execution paths.

To overcome this problem, we reduce the execution paths by applying several optimizations illustrated in Figure 13. First, given a basic block, if none of its instructions (or its descendant nodes' instructions) is data/control dependent on the service input, and it is not dominated by any permission, then it is ignored by Invetter. For example, we do not analyze node C in Figure 13. Second, if a basic block is dominated by a system level *privileged* permission validation, Invetter ignores this node. For example, node A in Figure 13 will not be further analyzed. We manually checked the logic of 21 interfaces checking privileged permissions, and found that all of them rely on the secure input provided by the system (e.g. uid from system interface *Binder.getCallingUid()*). Thus, this optimization do not introduce false negatives to Invetter. Besides, all normal conditional jumps recognized in §4 are also ignored in our analysis. After the optimizations, we obtain a simplified control flow graph which contains less paths to be analyzed.

6 EVALUATION

In this section, we evaluate Invetter's effectiveness, efficiency, and accuracy by applying it to 8 different Android system images, including 4 versions of AOSP (5.0, 6.0, 7.0, 7.1), and 4 system images customized by 3 different vendors (Samsung S6, XiaoMi Note2, HuaWei P9, and HuaWei Mate9). Additionally, since Android 8.0 utilizes a new dex file format, called *vdex*, and currently no tool can extract Java byte code from *vdex* files, thus we cannot apply Invetter to this version of Android. However, some vulnerabilities reported on the other Android versions still affect Android AOSP 8.0 and some other 3rd party Android images including Xiaomi Mix2(Android 8.0) and Huawei P10(Android 8.0). When applicable, we test our exploit programs against Android 8.0.

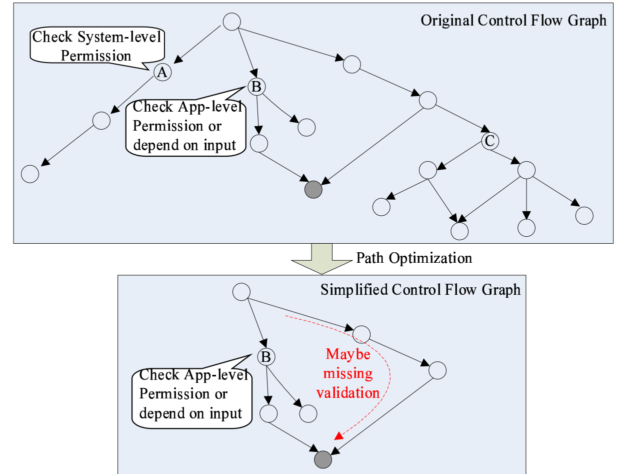


Figure 13: An example of optimizations for path-sensitive analysis.

Android Image (Version)	#Service	#Public interfaces	#Classes
AOSP(7.1)	(105)118	2126	19425
AOSP(7.0)	(103)115	2072	15524
AOSP(6.0)	(89)103	1786	15166
AOSP(5.0)	(87)96	1562	12179
Huawei Mate9(7.0)	(131)156	2292	20100
Huawei P9(7.0)	(118)139	1756	18608
XiaoMi Note2(6.0)	(100)126	2077	21961
Samsung S6(5.0)	(191)214	3584	18933

Table 1: The statistics of system services in different Android system images. In the second column, the number in the parentheses are system services covered by Kratos [26] and the other number is Invetter's result.

We also present some vulnerabilities identified by our approach as case studies. Our static analysis framework is running on a CentOS 7 server, with four 8-core 2.0GHz CPUs and 192 GB memory.

Statistics of analysis target. Our 8 target Android system images are summarized in Table 1. As we can see, the number of system services increases dramatically from Android 5.0 to 7.1, which leads to an increasing demand of security validations. Additionally, though both Huawei P9 and Huawei Mate 9 are based on the same Android version (7.0) and come from the same vendor, they have different numbers of services. It indicates the level of service customizations is fairly intense; even a same vendor may need to distinguish their own products. Moreover, standing out from other vendors, Samsung adds the most number of new services and public interfaces.

To illustrate the effectiveness of Invetter to extract system services, we also compare the number of system services covered by Invetter to the number reported in Kratos [26]. As presented in Table 1, by applying our new method to find system services through the client-side code (discussed in §5), Invetter covers more services than Kratos. After manually verifying these newly found services,

we confirm that all these services are valid Android system services and should be included in our analysis.

Efficiency. To illustrate the efficiency of Invetter, we summarize Invetter's analysis time on different Android images. For a specific Android image, Invetter needs about 85 minutes to generate an analysis report. Besides, the analysis time of Invetter is mostly consumed in the structure analysis phase, which applies an inter-procedure path-sensitive data-flow analysis. Since the whole analysis process can be finished in 11.8 hours for 8 Android images, we consider the execution time of our tool acceptable.

6.1 Tool Accuracy

After recognizing access controls used in the 8 tested system images, Invetter finds 1865 input validations used in Android framework (only 643 of them (34.48%) are protected by app-level permissions, and the remaining can be exploited without permission granted). We randomly select 800 (100 each for 8 system images) of them and find 71 false positives by manually checking. After manual inspection, we find that not all branches which return constants belong to input validations (e.g., some hard code constant returns). Since our tool recognizes all constant-returning branches as input validation, we mistakenly report such cases as input validations. From the 1865 input validations, Invetter finds 749 sensitive input validations after learning.

After the phase of vulnerability discovery, Invetter locates 103 possibly insecure ones in total by searching for the patterns discussed in §3. The results are shown in Table 2. We manually verify these insecure access controls, and find that among these reports, 86 are true positives. Some seemingly sensitive input validations in the end do not yield any sensitive subsequent actions (e.g., returning a true/false status). Unfortunately, it is extremely challenging to evaluate the completeness (i.e., false negatives) of our approach because the codebase of Android framework is too huge to inspect manually (more than 100,000 conditional branches). This is a common limitation of similar static analysis tools (e.g. Kratos [26], AceDroid [33]). As an empirical evidence from a small scale experiment on 5 Android services (including StatusBarManagerService, MmsServiceBrokers$BinderService, LocationManagerService, TextServiceManagerService, MediaSessionService$SessionManagerImpl) in AOSP 7.1, we manually identify their sensitive input validations as well as insecure input validations. These results are all successfully identified by Invetter. Thus, we believe that the coverage is decent.

6.2 Categorization of Identified Input Validations

To better understand the input validations of Android framework and the validations applied to the Android SDK, we conduct a measurement study in this section of all the manually checked input validations. Table 3 illustrates their distributions. About 36% of input validations in Android framework and 12% of validations in Android SDK are used to verify the caller's identities such as uid, package name, or whether it holds a critical permission. Most of these validations are critical security checks, thus bypassing them may cause serious consequences. Besides, about 10% of the input validations in Android framework and 31% of validations in Android SDK are designed to restrict the usage of sensitive system resources. For example, check whether the type of a given message is permitted. Bypassing these checks can also lead to security flaws, although less likely compared to the identity checks. Thus, in total, more than 40% of the input validations in Android are used to ensure the secure usage of sensitive resources.

Android Image (Version)	C1	C2	C3
AOSP(7.1)	27	8	-
AOSP(7.0)	24	7	-
AOSP(6.0)	23	6	-
AOSP(5.0)	20	5	-
Huawei Mate9(7.0)	27	5	9
Huawei P9(7.0)	25	5	26
XiaoMi Note2(6.0)	28	7	14
Samsung S6(5.0)	35	6	41

Table 2: The number of possible insecure validations in different Android images. These results are categorized by: incorrectly trusting app-supplied data(C1), incorrectly trusting code in the app process(C2), and weakened validation in customized system services(C3).

Category	Android Framework	Android SDK
Verify caller identity	189	30
Restrict usage of sensitive resources	50	72
Security irrelevant validations	258	130
Total	497	232

Table 3: The categorization of input validations.

6.3 Tool Effectiveness

From all the 86 identified insecure sensitive input validations (true positives mentioned earlier), we further hope to understand whether these cases are actually exploitable. For our purposes, we manually investigated them and indeed confirm there exist a large number of exploitable vulnerabilities. Admittedly we may not have done an extensive job in analyzing these cases, and there may be cases that are difficult to trigger but can become exploitable with more efforts. Therefore our estimate of exploitable vulnerabilities is only a lower bound.

After our analysis, we confirm at least 20 exploitable vulnerabilities, presented in Table 4. They range from privilege escalation, privacy leakage, to clearance of system files, etc. Among them, 11 input validations incorrectly check the caller's identity using app-supplied data. One of them is illustrated in §3, as shown in Figure 5. Another example is that an app-supplied *userId* is used to verify the identity of the caller. Furthermore, for one case, we find a counterpart of native service which is properly protected, while its Java-level wrapper service is left unprotected. A regular app directly accessing the native service will be denied, yet accessing the Java service allows indirect access to the native service, effectively a confused deputy example.

Class Name	Affected Frameworks											Attack	Detail	Vendor Reply
	AOSP					Third Part Rom								
	5.0	6.0	7.0	7.1	8.0	XM N2	XM M2	HW M9	HW P9	HW P10	SU S6			
AccessibilityManagerService	•	•	•	•	•	•	•	•	•	•	•	A1	interrupt all accessibility services	N
NetworkManagerService			•	•	•		•	•		•		A1	modify VPN configurations	N
AccessibilityManager	○	○	○	○	○	○	○	○	○	○	○	A1	expose all hidden interfaces to user	N
Window ManagerService	•	•	•					•	•		•	A1	create phishing toast window	F
AccessibilityManagerService	•	•	•	•	•	•	•	•	•	•	•	A1	send arbitrary accessibility event	N
InputManagerService	•	•	•	•	•	•	•	•	•	•	•	A1	send crafted physical key event	N
MediaSessionService	•	•	•	•	•	•	•	•	•	•	•	A1	send crafted media key event	N
DropBoxManagerService	•	•	•	•	•	•	•	•	•	•	•	A2	clear kernel logs	N
Atfwd#	•	•	•									A1	send arbitrary keyword/touch event, erase sdcard content,etc.	N
CNEService#	•	•	•									A1	modify the wifi spot connection policy	N
MiuiInitServer						⊗	⊗					A1	do factory reset for the pre-install apps	C
AudioServer						⊗						A1	mute the device	F
AudioServer						⊗	⊗					A1	add or remove bluetooth device	C
WhetstoneActivity ManagerService						⊗						A1	modify system white list	N
Regionalization Server						⊗	⊗					A1	delete arbitrary file under system dir	C
HwAttestationServer								•	•			A1	obtain the unique id of mobile device	F
HwPhoneServer								⊗		⊗		A3	obtain the cell location of mobile device	C
HwAttestationServer								⊗	⊗	⊗		A3	obtain the public keys stored on the device	C
HwSysResManagerService									•	•		A1	allocate arbitrary memory	C
Device*											⊗	A1	store arbitrary MMS on the device	C

Table 4: The exploitable vulnerabilities exposed by Invetter. We show the effect of these vulnerabilities in different versions of AOSP, as well as XiaoMi Note2(XMN2, Android6.0), XiaoMi Mix2(XMM2, Android8.0), HuaWei Mate9(HWM9, Androi7.0), HuaWei P9(HWP9, Android6.0), HuaWei P10(HWP10, Android8.0), and Samsung S6(SUS6, Android5.0). These vulnerabilities can be categorized by: incorrectly trusting app-supplied data(•), incorrectly trusting code in the app process(○) and weakened validation in customized system services(⊗). By exploiting these vulnerabilities, attackers can conduct privilege escalation attacks(A1), log overflow attacks(A2), and private leakage attacks(A3). The 5th column provides the details about each vulnerability. The row labeled with * can also be located by Kratos [26]. Since Atfwd(labeled with #) is hidden by the SEAndroid policy, its vulnerabilities can be exploited only if SEAndroid is disabled or 3rd-party vendor modifies the policy. We submitted these vulnerabilities to Google and other corresponding vendors. The last column lists the status, with C stands for this vulnerability has been confirmed, N stands for we have notified them and currently not received their responses, and F stands for it has been fixed in the latest version of Android image.

Besides, one access control is misplaced only in the Android SDK (and not in Android services). Interestingly, there are 4 other similar cases that do not seem exploitable at the moment but nevertheless it is a potential problem.

Finally, we find all four studied customized system images weaken the security enforcements when they modify old or add new services, resulting in 10 exploitable vulnerabilities. In fact, we find that in most of the cases, vendors barely put any security checks in their new service code, suggesting that third-party vendors are less security-conscious overall compared to Google.

6.4 Case Study

We now choose a subset of the 20 cases to explain how the vulnerabilities manifest themselves and how they can be exploited. For interested readers, our anonymous demonstration video can be found at https://youtu.be/erLY_OMi4kQ. We are in the process of responsibly disclosing the details to Google and other related third-party vendors.

Hidden interfaces left by the microchip manufacturer (privilege escalation). Atfwd is a system app provided by the microchip manufacturer Qualcomm, and pre-installed on many Qualcomm-based Android devices. Atfwd registers a system service called AtCmdFwd, which accepts various commands through app-accessible interfaces. Specifically, the commands accepted by AtCmdFwd are illustrated in Table 5. Although it is designed to reject commands from non-system apps, we show that due to an insecure input validation, a malicious app can fake its identify. As a result, malware can arbitrarily inject commands such as push physical buttons, or trigger motions on the touch screen. Additionally, we notice that due to a similar insecure input validation, AtCmdFwd exposes some sensitive system operations, for example, erase the external/internal storage or reboot/shutdown the device. Surprisingly, we are unable to locate the user of these exported interfaces and unsure why they are pre-installed everywhere. Luckily, in recent updates of SEAndroid policies, Atfwd becomes inaccessible to regular apps. However, its vulnerabilities can be exploited

Command	Event Handler	Description
+CKPD	AtCkpdCmd	Send an arbitrary key/button press event
+CTSA	AtCtsaCmd	Send a touch screen motion event
+CFUN	AtCfunCmd	Reboot the device
+CRSL	AtCrslCmd	Set audio stream volume
+CMAR	AtCmarCmd	Erase the external/internal storage
CSS	AtCssCmd	Get default display settings
$QCPWRDN	AtQcpwrdnCmd	Shutdown the device

Table 5: Commands accepted by AtCmdFwd. The first column shows the command tokens accepted by the system service, and the second column presents the corresponding event handler triggered by the commands. The final column describes the effect of each command.

if SEAndroid is disabled (in a lower version of Android), if a 3rd-party vendor misconfigures the policy, or if an unrelated system process is compromised first which can then reach the service. Our demonstration video shows that the interfaces can be utilized in a zero-permission app.

Sending arbitrary accessibility event (privilege escalation). Accessibility service is commonly registered to *AccessibilityManagerService* by apps, providing convenience to assist the mobile user's operations, such as auto-filling data (e.g., password) or touching a point on the screen. Although accessibility services are originally designed to assist users with disabilities, it is not limited to this purpose. For example, many UI testing frameworks use accessibility service to gain access to specific app views, such as the *uiautomater* in Android framework. Besides, some apps use accessibility services to provide sensitive functionalities, such as reading content of user's current view, or alerting the user.

Interestingly, we find the input validation used in *AccessibilityManagerService* is vulnerable. By exploiting this vulnerability, a malicious app can deliver arbitrary accessibility events to any targeted accessibility service. For example, we can target Notification Check [7], a popular app used to manage various notifications on your phone. This app registers an Accessibility Service, which allows it to listen for the arrival of notifications, i.e., accessibility events with type *TYPE_NOTIFICATION_STATE_CHANGED* dispatched by *AccessibilityManagerService*. With the event injection capability, a malicious app can deliver crafted events to *Notification Check* for phishing. Likewise, by sending a forged event to the accessibility service which auto-fills user password, malware can steal user password stored in this app, causing severe information leakage. To emphasize, as the vulnerability lies in the system framework *AccessibilityManagerService*, any app that registers their app-specific service with it can become vulnerable. We tested the vulnerability in the latest Android (8.0) and 3rd party Android images. We confirm that it is still present.

Stealthy phishing attack (privilege escalation). The Android OS provides a convenient functionality for developers to popup a message on the screen, called *Toast*. When a toast is displayed, it only fills the amount of space required for the message and the current top activity remains visible and interactive. Originally, the layout of Toast window is fixed (like a notification to user) and cannot be customized by apps. However, Invetter finds an interface in *WindowManagerService* which allows a malware to create crafted *Toast* message with arbitrary scope of view space. As a result, a malware can completely customize the toast window (e.g., a transparent *TextField* that captures the user input), and display it on top of an arbitrary app. This is because there are two separated paths based on different inputs that can popup a toast window. One path requires the caller must have a *SYSTEM_ALERT_WINDOW* permission. However, the other one does not apply any validation, leading to the vulnerability introduced above. We confirm this vulnerability with an exploitation on Nexus 6 (AOSP 7.0), which can popup a phishing window without the user noticing.

Controlling the media player (privilege escalation). *MediaSessionService* provides a method named *dispathchMediaKeyEvent* that allows apps to send out media key events to control the current running media player, such as stopping a media file or playing another. This method is originally designed as a hidden method since it is labeled as *@hide* in Android SDK. Normally, a developer can not call this method in his app. However, since the Android SDK is executed in app's process, an app can overwrite the manager side RPC code, and invoke this method anyways by creating its own media key event. *MediaSessionService* conducts a verification to make sure the input key event is a kind of media key events, after that it clears the caller's identity in Binder by calling *clearCallingIdentity*, which means that the sender of the media key event is erroneously set to system. This insecure validation allows an attacker to create various kinds of media key events to control the current running media player. As an exploitation experiment, we select two popular media players in China, *NetEaseMusic* and *QQMusic* as targets. Both of them can be controlled by the malicious app we developed.

Forcing factory reset (privilege escalation). In the system image of XiaoMi Note 2, Invetter discovers a sensitive service interface, called *doFactoryReset*, which is not protected by any access controls. This method resides in the customized system service *MiuiInitServer*. Doing factory reset is a system level behavior that should be protected with critical enforcements, and commonly, it can only be accessed by pre-installed system apps. Actually, AOSP has a similar method called *factoryReset*, which requires a *privileged* permission (*CONNECTIVITY_INTERNAL*). However, Invetter finds that no check is performed in XiaoMi's system image and any app can access it without any restriction. In this case, we identify a weakened access control in the newly added system services by a third-party vendor, which demonstrates that customized system images may weaken the original security enforcements of Android.

Clearing Android Kernel Log (Log overflow). *DropBoxManagerService*(DBMS) is a persistent, system-wide, blob-oriented logging service of Android (not to be confused with the file sharing app which is also called *Dropbox*). Commonly it is used for recording chunks of data from various sources, such as application

crashes, kernel log records, etc. Invetter reports a public interface, *add*, in this service which does not enforce any permission check. It only conducts sensitive input validation based on the untrusted app-supplied data. This makes it possible for a malicious app to access this interface, although it is designed for system only. The app can fake the log information to mislead security analysts who use the log reports, or even can erase the original system logs with fake data. This is because DBMS uses a fixed-length queue to manage logs in a system directory, and old data is discarded directly when the maximal size reaches.

Deleting system files (privilege escalation). To prevent less privileged apps from accessing files stored by high privileged system/app processes, the Android sandbox separately stores app files in their own app's directory. As a result, only privileged apps or system can access the sensitive resources. However, Invetter reports a unprotected public interface, called *deleteFileUnderDir*, in a customized system service *RegionalizationService* from XiaoMi Note 2. Using this interface, the caller can delete arbitrary files owned by the current running process. Since this system service is executed in the system process, the caller client can delete system files by calling it. Since this critical interface is not protected by any permission or secure input validation, malicious apps can trigger the file deletion whenever the service is running. This vulnerability is not acknowledged in our testing device due to an incompatibility, but confirmed by our in-depth code review.

7 DISCUSSION

Native code. Since Invetter is implemented based on Soot, which cannot analyze the native code of Android, Invetter currently cannot find vulnerabilities inside Android native services. We manually checked the native services in Android framework, and find that only 15 services are not analyzed by Invetter, e.g., Camera Service. Since the code base of these services are relatively low, we believe the impact is small because Invetter can find most system services in the Android framework. Besides, as discussed in §5, we proposed an approach to find Java byte code clients of services, including the native ones. Thus, although we still cannot analyze the native services, we can analyze and find insecure code within the Java clients of native services, which cannot be achieved by the existing approaches.

Inferring sensitive inputs. It is an open problem to automatically infer sensitive inputs crossing a trust boundary in any large software (e.g., user-to-kernel and app-to-service). However, in more limited scenarios, inferring sensitive data has been considered and studied using various techniques. For example, TaintDroid [8] labels the return value of a hand-curated list of Android APIs as sensitive (e.g., `getLastKnownLocation()`. UIPicker [21] and SUPOR [16] use learning-based approaches to identify sensitive inputs through UI. Similar to our idea, they first manually label some sensitive UI elements (e.g., input boxes) and then use machine learning to infer other sensitive ones via co-location analysis. Unlike UI elements, the scale and complexity of sensitive inputs in programs are much more challenging. Specifically, we are not aware of any good learning strategies (e.g., by co-location or co-occurrence) that can be generally applied. This is why instead of learning "sensitive inputs", we choose to learn "sensitive input validations" — as the latter can be learned by co-occurrence.

Recommendations for implementing secure validations. This paper reveals vulnerabilities caused by insecure input validations. By comparing insecure and secure input validation implementations, we recommend that sensitive input validation should be performed in the following way: First, all the data derived from Android apps, including Android SDK, should not be trusted. To validate the app identity, system controlled app signatures (e.g. information managed by the Binder mechanism) should be used. Besides, any system level access controls should not be placed in user apps or Android SDK. Then, vendor customization should be more careful when modifying system services, so that not to remove sensitive input validation.

8 RELATED WORK

In this section, we review related prior research and compare our work with those studies.

Vulnerability detection in Android. The problem of security vulnerabilities in Android has been extensively studied. Unixdomain [27] and ION [35] study the Android socket and low-level heap interfaces, and report unprotected public interfaces by finding missing permission validations. ASV [15] discovered a design trait in the concurrency control mechanism of Android system server, which may be vulnerable to DOS attacks. Besides, IntentScope [17] shows that some Android components, e.g. services, accept inter-component access from other components, e.g. apps, and because some components mis-configured their intent filters, they can be accessed by unauthorized apps. This paper also discuss access control of Android framework. Moreover, Zhang [37] shows that in Android, after the app is uninstalled, some app data is not completely removed, causing privacy leakage. These works focus on a specific vulnerability pattern, e.g. concurrency bugs. Different from the work above, this paper focuses on input validation problems in Android framework.

Additionally, some works focus on discussing the explicit permission based access control mechanism of Android. Felt [10] shows that pre-installed apps can access critical system resources, meanwhile they may open interfaces that accept requests from low privileged apps. Since the access control in these apps may be weak, low privileged malicious apps can utilize them as a step stone to access high privileged resources. Kratos [26] compares the permission enforcement along different calling stacks of Android system services, and finds vulnerabilities ranging from privilege escalation to DoS. Besides, AceDroid [33] focuses on the inconsistent permission enforcement introduced by different vendors. Similar to our purpose, buzzer [6] also aims to find incorrect input validations in Android services, but unfortunately, we find that most of their works are done manually, and only several vulnerabilities which crash the services can be detected automatically. Gu, et al. [31] does not formally define sensitive input validations and relies on a set of manually-created lists of sensitive APIs. In addition, their system reports 22 vulnerabilities but only 3 are related to incorrect sensitive input validations (many are repeated invocations of APIs that crash the system which is beyond our scope).

By carefully studying the 3 reported input validations, we notice that 2 of them are also located by Invetter (one is listed in Table 4, and the other is discarded during manual verification because of the low severeness), and the remaining one was already fixed by Google. In this paper, we discuss incorrect security forced input validations in Android services, and weakened access controls in customized Android images. Our systematical approach reveals 86 vulnerabilities, among which 20 are confirmed exploitable, from 8 Android images. To the best of our knowledge, they are not systematically studied in existing work.

Weakened access controls in Android customizations. The security risks introduced by the customization of Android system images are also studied before. Prior researches [1, 9, 10] focus on the pre-installed apps in Android factory images and report the presence of several kinds of the vulnerabilities, such as over-privileged, permission re-delegation, hanging attribute references, etc. Unlike these works, this paper focuses on the vulnerabilities inside Android system services.

Other related studies [2, 11, 14, 30] find that customized system images modify security configurations, and incorrect modifications bring in security vulnerabilities. Like these papers, we also discuss weakened security enforcement in customized images. However, we focus on identifying insecure validations inside Android system services, which requires deep understanding of the service code.

Static analysis on Android. We detect the vulnerabilities in Android framework by using static analysis. Techniques serving the similar purpose have been extensively studied [2, 4, 5, 26]. As one of the most popular techniques used to analyze Android framework as well as apps, static taint analysis monitors the data propagation along Android framework as well as apps [3, 12, 18, 23, 29, 34]. They answer the question of what data(source) flows into what destination(sink). Their use of static analysis techniques and source/sink flow control is an important concept in our work. We use static taint analysis to track the propagation of service input. PScout [4] and Axplorer [5] use static analysis to enumerate all permission checks in the Android framework and map all the permission usage to the corresponding system methods. Although they reveal what permissions are enforced in a given method, they cannot find missing validations. A similar work is Kratos, which finds missing security validations by comparing permission enforcement along different calling stacks. However, since its analysis is built on top of the call graph, it cannot find finer-grained inconsistencies which can be revealed in this paper. Besides, none of the above approaches can identify incorrect sensitive input validation.

Other input validations. The traditional input validation studies mainly focus on the web apps (SQL injection) and programs that are not memory-safe (C/C++ programs and OS kernel). For example, Mokhov, et al [20] studies the vulnerabilities in Linux kernel, with regards to two input validation errors: buffer overflow and boundary condition error. Scholte, et al. [25] studies the evolution of input validations vulnerabilities in web apps. They find that these vulnerabilities have not changed significantly and most of them result from the missing check of structural input strings. Yamaguchi, et al. [32] uses code property graph to characterize known vulnerability types in Linux kernel, such as buffer overflow, integer overflow, format string vulnerability and memory corruption. In particular, these vulnerabilities rely on a small set of well-defined sensitive input (e.g. user-space pointers) as input to static taint analysis. However, our paper focuses on the opposite end of the spectrum where we are not even clear what input should be considered sensitive. Different from the above studies that focus on well-known sensitive input, our paper focuses on the sensitive input validations in Android system services, which are unstructured, ill-defined and fragmented.

9 CONCLUSION

In this work, we make the first attempt to systematically study the input validations used in Android framework. We propose Invetter, a static analysis framework which focuses on the sensitive input validations in the Android framework and in the customized third-party system services. We demonstrate the effectiveness of our approach by applying Invetter on 4 versions of Android AOSP framework, and another 4 third-party vendors' Android images. Finally, Invetter reports 20 exploitable vulnerabilities which can lead to various kinds of attacks, such as privilege escalation and privacy leakage. Actually, most of these attacks do not require any permission and can be conducted by any app installed on the victim's mobile device. Our findings show that a critical way to well implement the access control should both consider the Android permission and the inputs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U1636204, 61602121, U1736208, 61602123), the National Program on Key Basic Research (NO. 2015CB358800). Yuan Zhang was supported in part by the Shanghai Sailing Program under Grant 16YF1400800. Min Yang is corresponding author of Shanghai Institute of Intelligent Electronics & Systems, and Shanghai Institute for Advanced Communication and Data Science, and was supported in part by the K.C.Wong education foundation, Hong Kong.

REFERENCES

[1] Aafer, Y., Zhang, N., Zhang, Z., Zhang, X., Chen, K., Wang, X., Zhou, X., Du, W., and Grace, M. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *CCS* (2015).
[2] Aafer, Y., Zhang, X., and Du, W. Harvesting inconsistent security configurations in custom android roms via differential analysis. In *USENIX* (2016).
[3] Arzt, S., Rasthofer, S., Fritz, C., SPRIDE, E.-B.-E., Bartel, A., Klein, J., Traon, Y.-L., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices* (2014).
[4] Au, K.-W.-Y., Zhou, Y., Huang, Z., and Lie, D. Pscout: analyzing the android permission specification. In *CCS* (2012).
[5] Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. On demystifying the android application framework: Re-visiting android permission specification analysis. In *USENIX* (2016).
[6] Cao, C., Gao, N., and Peng, L. Towards analyzing the input validation vulnerabilities associated with android system services. In *ACSAC* (2015).
[7] Durkin, S. Notification check, 2014. https://play.google.com/store/apps/details?id=com.sndurkin.notificationcheck.

[8] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).

[9] Felt, A.-P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *CCS* (2011).

[10] Felt, A.-P., Wang, H. J., and Moshchuk, A. Permission re-delegation: Attacks and defenses. In *USENIX* (2011).

[11] Gallo, R., Hongo, P., and Dahab, R. Security and system architecture: Comparison of android customizations. In *WISEC* (2015).

[12] Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust* (2012).

[13] Google. Operating system market share, 2018. https://developer.android.com/guide/topics/permissions/index.html.

[14] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock android smartphones. In *NDSS* (2012).

[15] Huang, H., Zhu, S., Chen, K., and Liu, P. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *CCS* (2015).

[16] Huang J, Li Z, X. X. e. a. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX* (2015).

[17] Jing, Y., Ahn, G.-J., Doupe, A., and Yi, J.-H. Checking intent-based communication in android with intent space analysis. In *CCS* (2016).

[18] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *CCS* (2012).

[19] Miller, G. A. Wordnet: A lexical database for english. In *Communications of the ACM* (1995).

[20] Mokhov S A, Laverdiere M A, B. D. Taxonomy of linux kernel vulnerability solutions. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education. Springer* (2008).

[21] Nan Y, Yang M, Y. Z. e. a. Uipicker: User-input privacy identification in mobile applications. In *USENIX* (2015).

[22] NetMarketShare. Operating system market share, 2017. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1&qpsp=2017&qpnp=1 &qptimeframe=Y.

[23] Octeau, D., McDaniel, P., Jha, S., Bartel, A., and Bodden, E. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX* (2013).

[24] research group, S. Soot, 2017. https://github.com/Sable/soot.

[25] Scholte T, Balzarotti D, K. E. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *International Conference on Financial Cryptography and Data Security. Springer* (2011).

[26] Shao, Y., Ott, J., Chen, Q. A., Qian, Z., and Mao, Z. M. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS* (2016).

[27] Shao, Y., Ott, J., Jia, Y.-J., Qian, Z., and Mao, Z. The misuse of android unix domain sockets and security implications. In *CCS* (2016).

[28] Tan, P.-N., Michael, S., and Kumar, V. Introduction to data mining. In *Addison-Wesley* (2005).

[29] Wei, F., Roy, S., Ou, X., et al. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *CCS* (2014).

[30] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The impact of vendor customizations on android security. In *CCS* (2013).

[31] Yacong, G., Yao, C., and LingYun, Y. Exploiting android system services through bypassing service helpers. In *SecureComm* (2016).

[32] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 590–604.

[33] Yousra, A., Jianjun, H., and Yi, S. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *NDSS* (2018).

[34] Yuhong, N., Zhemin, Y., and XiaoFeng, W. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *NDSS* (2018).

[35] Zhang, H., She, D., and Qian, Z. Android ion hazard: The curse of customizable memory management system. In *CCS* (2016).

[36] Zhang, M., and Duan, Y. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS* (2014).

[37] Zhang, X., Ying, K., Aafer, Y., Zhenshen, Q., and Wenliang, D. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS* (2016).