Understanding and Defending the Binder Attack Surface In Android

Huan Feng, and Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
{huanfeng, kgshin}@umich.edu

ABSTRACT

In Android, communications between apps and system services are supported by a transaction-based Inter-Process Communication (IPC) mechanism. `Binder`, as the cornerstone of this IPC mechanism, separates two communicating parties as client and server. As with any client–server model, the server should not make any assumption on the validity (sanity) of client-side transaction. To our surprise, we find this principle has frequently been overlooked in the implementation of Android system services. In this paper, we try to answer why developers keep making this seemingly simple mistake by studying more than 100 vulnerabilities on this attack surface. We analyzed these vulnerabilities to find that most of them are rooted at a common confusion of where the actual security boundary is among system developers. We thus highlight the deficiency of testing only on client-side public APIs and argue for the necessity of testing and protection on the `Binder` interface — the actual security boundary. Specifically, we design and implement *BinderCracker*, an automatic testing framework that supports context-aware fuzzing and actively manages the dependency between transactions. It does not require the source codes of the component under test, is compatible with services in different layers, and performs 7x better than simple black-box fuzzing. We also call attention to the attack attribution problem for IPC-based attacks. The lack of OS-level support makes it very difficult to identify the culprit apps even for developers with adb access. We address this issue by providing an informative runtime diagnostic tool that tracks the origin, schema, content, and parsing details of each failed transaction. This brings transparency into the IPC process and provides an essential step for other in-depth analysis or forensics.

1. INTRODUCTION

Android is the most popular smartphone OS and dominates the global market with a share of more than 82% [36]. By the end of 2015, the total number of Android devices surpassed 1.4 billion, and more than 1.6 million mobile apps were available in Google Play for download [16, 28]. The developers of these apps are not always trustworthy; many of them might be inexperienced, careless or even malicious. Therefore, proper isolation between apps and the system is essential for robustness and security.

To meet this requirement, apps in Android execute in the application sandboxes. They depend on Inter-Process Communications (IPCs) extensively to interact with the system and other apps. `Binder`, as the cornerstone of this IPC mechanism, has long been believed as one of the most secure/robust components in Android. However, during the past year, there have been multiple CVE (Common Vulnerabilities and Exposures) reports discussing attacks exploiting the `Binder` interface [6–9, 29]. Interestingly, none of these attacks tries to undermine the security of `Binder` driver, but instead use `Binder` only as an attack gateway (entry point). A careful examination of the attack surface has led us to the discovery of the fundamental cause of this attack vector: an attacker can directly inject faulty transactions into system services by manipulating the `Binder` interface, and hence bypass all client-side sanity checks. Theoretically, this should not be an issue — a system service should not hinge on the validity of client-side transactions, and should always be robust on its own. However, we found that this principle has frequently been overlooked in the implementation of many Android system services, which led us to the following questions: *why system developers keep making this seemingly simple oversight, and what can we do to help mitigate this problem?*

To answer these questions, we conduct the first in-depth analysis of this attack surface. Specifically, we studied more than 98 generic system services (by Google) and 72 vendor-specific services (by Samsung) in 6 Android versions, and identified 137 vulnerabilities (already de-duplicated across versions) on this surface. We analyzed 115 of them in Android source codes and found that sanity checks are most extensive around client-side public APIs, and quickly become sporadic/careless after this defense line. Specifically, RPC parameters that are not exposed via public APIs are frequently left unchecked and the underlying (de-)serialization process of these parameters is often unprotected. This suggests that there is a mis-conception of where the security boundary is for Android system services — many seem to assume the security/trust boundary to be at the client-side public APIs, and whatever happens thereafter is free from obstruction since they already belong to the system territory. This mis-conception is understandable since Android

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA
© 2016 ACM. ISBN 978-1-4503-4771-6/16/12. . . $15.00
DOI: http://dx.doi.org/10.1145/2991079.2991120

provides convenient and automatic code generation tools (AIDL) that at one side relieve the developers from writing their own IPC stack, but at the other side hide all the details about RPC and `Binder`. Thus, we argue for the necessity of introducing automatic testing and protection at the `Binder` surface, i.e., the actual security boundary.

All the vulnerabilities reported in this paper are identified by *BinderCracker*, a precautionary testing framework we developed for `Binder`-based RPCs. *BinderCracker* is a context-aware fuzzing framework that understands the input and output structure of each RPC transaction as well as the inter-dependencies between them. This is essential because many transactions require inputs of remote object handles which are output of other transactions and cannot be recorded in the form of raw bytes. Before fuzzing a transaction, *BinderCracker* will automatically replay all the transactions it depends on and generate the correct context. *BinderCracker* does not require source codes of the services under test and works for services in both the Java and native layers. Thus, it is readily compatible with both Android system services and vendor-specific services. *BinderCracker* achieves effective vulnerability discovery — it identified 7x more vulnerabilities than simple black-box fuzzing within the same amount of time. Furthermore, since *BinderCracker* understands the schema of each low-level RPC transaction, we can easily configure it to test high-level abstraction or protocol built on top of the `Binder` primitives, such as Intent communications or app-specific protocols.

To help mitigate this emerging attack surface, we need to eliminate potential vulnerabilities as early as possible in the development cycle. Specifically, we suggest the use of various precautionary testing techniques (including *BinderCracker*) before each product release. This can stop a large number of vulnerabilities from reaching the end-users. In fact, many severe vulnerabilities [6–9, 29] could have been avoided had *BinderCracker* been deployed. Notably, 60% of the vulnerabilities identified by *BinderCracker* still remain unfixed. We summarized these vulnerabilities and have already reported them to AOSP. Many of the vulnerabilities we identified are found to be able to crash the entire Android Runtime, while others can cause specific system services or system apps to fail. Some vulnerabilities have further security implications, and may result in permission leakage, privileged code execution, targeted or permanent Denial-of-Service (DoS).

In case vulnerabilities leak through precautionary testing into the deployment phase, we need runtime defenses on this attack surface. Here, we addressed the urgent problem of attack attribution for IPC-based attacks, which have not received enough attention from the security community. Due to the lack of OS-level support, it is extremely difficult to identify the culprit app even for developers with adb access, let alone for average users. This suggests that an attacker app can sabotage the system or crash other apps without being accused of, or may even blame it on others. For example, the attacker app can crash Android Runtime whenever the user opens a competitor app, creating the illusion that the competitor app is buggy. Similar attacks are not rare between businesses with close competition [37]. We addressed this issue by building an informative runtime diagnostic tool. It maintains the sender, schema, content and parsing information for each ongoing transaction, in case a failure/attack happens. Whenever a system service fails when processing an incoming transaction, a detailed report with the transaction information will be generated and a visual warning will be prompted to the user. The reporting process can also be triggered by access to privileged APIs or abnormal permission request, to catch attacks that do not warrant a program crash. This brings transparency into IPC communications and constitutes an essential first step for other in-depth analysis or forensics.

This paper makes the following contributions: we

- Provide a systematic analysis of the attack surface by conducting security and root cause analysis on 100+ vulnerabilities. We summarized the common mistakes made by system developers and found the attack surface persists largely due to a common confusion of where the actual trust boundary is;

- Design and implement, *BinderCracker*, a context-aware fuzzing framework for Android IPCs that actively managed the dependencies between transactions. *BinderCracker* is compatible with Android system services, vendor-specific services and can be easily configured to fuzz high-level abstractions or protocols. It identifies 7x more vulnerabilities than a simple black-box fuzzing approach;

- Address the attack attribution problem for IPC-based attacks by building a system-level diagnostic tool. By tracking the origin, schema and content of ongoing transactions, it brings transparency into Android IPCs and provides an essential step towards in-depth runtime analysis and defense.

The rest of the paper is organized as follows. Section 2 summarizes related work in the field of software testing and Android security. Section 3 introduces `Binder` and AIDL in Android, and describes how Android uses these to build system services. Section 4 examines the attack surface and focus on explaining what mistakes have the developers made and why. Section 5 details the design and implementation of our testing framework, *BinderCracker*. Section 6 gives a comprehensive discussion on how to mitigate this attack surface with a special focus on the attack attribution problem. Section 7 discusses the insight and other potential use cases of *BinderCracker*, and finally, the paper concludes with Section 8.

2. RELATED WORK

Discussed below is related work in the field of software testing and Android security.

Software Testing. In the software community, robustness testing falls into two categories: *functional* and *exceptional* testing. Functional testing focuses on verifying the functionality of software using expected input, while exceptional testing tries to apply unexpected and faulty inputs to crash the system. Numerous efforts have been made in the software testing community to test the robustness of Android [1, 2, 19, 23, 26, 38]. Most of them focus on the functional testing of GUI elements [1, 2, 19, 23]. Some have conducted exceptional testing on the evolving public APIs [26]. In this paper, we highlight the deficiency of testing only on public APIs and conduct an exceptional testing on lower-level `Binder`-based RPC interfaces.

Android Security. Android has received significant attention from the research community as an open source operating system [4, 11, 12, 18, 25, 30, 34]. Existing Android security studies largely focus on the imperfection of high-level permission model [13, 14, 27], and the resulting issues, such as information leakage [11], privilege escalation [4, 30] and collusion [25]. Our work highlights the insufficient protection of Android's lower-level `Binder`-based RPC mechanism and how it affects the robustness of system services.

There also exist a few studies focusing on the IPC mechanism of Android [5, 10, 21, 24, 31]. However, they largely focus on one specific instance of Android IPC — Intent. Since the senders and recipients of Intents are both apps, manipulating Intents will not serve the purpose of exposing vulnerabilities in system services. Some researchers also provide recommendations for hardening Android IPCs [21, 24] and point out that the key issue in Intent communication is the lack of formal schema. We demonstrate that even for mechanisms enforcing a formal schema, such as AIDL, robustness remains as a critical issue. There have also been some parallel attempts on fuzzing the `Binder` interface in the industry [15, 17]. However, they focused on the technical details of implementing Proof-of-Concept (PoC) exploits on this interface and only tested simple fuzzing techniques. Our work instead, focuses on understanding the origin of the `Binder` attack surface and proposes practical defenses. We summarized the common mistakes made by system developers by studying 100+ real vulnerabilities and addressed the urgent problem of attack attribution for IPC-based attacks. Moreover, our context-aware fuzzing framework, *BinderCracker*, is sophisticated and performs much more effectively than a simple black-box fuzzing.

3. ANDROID IPC AND BINDER

Android executes apps and system services as different processes and enforces isolation between them. To enable different processes to exchange information with each other, Android provides, `Binder`, a secure and extensible IPC mechanism. Described below are the basic concepts in the `Binder` framework and an explanation of how a typical system service is built using these basic primitives.

3.1 Binder

In Android, `Binder` provides a message-based communication channel between two processes. It consists of (i) a kernel-level driver that achieves communication across process boundaries, (ii) a `Binder` library that uses `ioctl` syscall to talk with the kernel-level driver, and (iii) upper-level abstracts that utilize the `Binder` library. Conceptually, `Binder` takes a classical client–server architecture. A client can send a transaction to the remote server via the `Binder` framework and then retrieves its response. The parameters of the transaction are marshalled into a `Parcel` object which is a serializable data container. The `Parcel` object is sent through the `Binder` driver and then gets delivered to the server. The server de-serializes the parameters of the `Parcel` object, processes the transaction, and returns a response in a similar way back to the client. This allows a client to achieve *Remote Procedure Call* (RPC) and invoke methods on remote servers as if they were local. This `Binder`-based RPC is one of the most frequent forms of IPC in Android, and underpins the implementation of most system services.

```
interface IQueueService {
  boolean add(String name);
  String peek();
  String poll();
  String remove();
}
```

Figure 1: An example AIDL file which defines the interface of a service that implements a queue.

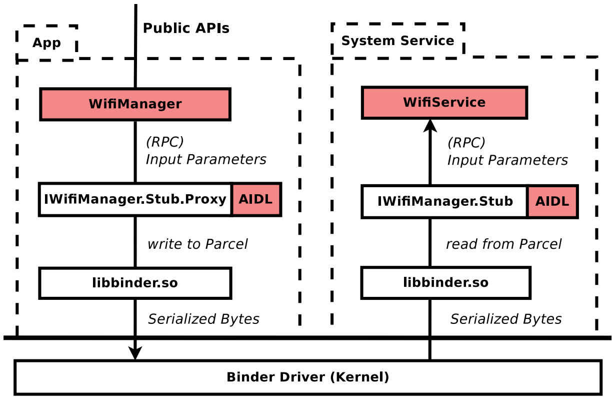


Figure 2: How does an app communicate with a system service using `Binder`-based RPC (using Wi-Fi service as an example)? The red shaded region represents the codes that need to be provided/implemented by the service developer.

3.2 AIDL

Many RPC systems use IDL (*Interface Description Language*) to define and restrict the format of a remote invocation [24], and so does Android. The AIDL (*Android Interface Description Language*) file allows the developer to define the RPC interface both the client and the server agree upon [3]. Android can automatically generate Stub and Proxy classes from an AIDL file and relieve the developers from (re-)implementing the low-level details to cope with native `Binder` libraries. The auto-generated Stub and Proxy classes will ensure that the declared list of parameters will be properly serialized, sent, received, and de-serialized. The developer only needs to provide a `.aidl` file and implement the corresponding interface. In other words, the AIDL file serves as an explicit contract between client and server. This enforcement makes the `Binder` framework extensible, usable, and robust. Fig. 1 shows an example AIDL file that defines the interface of a service that implements a queue.

3.3 System Service

We now describe how the low-level concepts in the `Binder` framework are structured to deliver a system service, using Wi-Fi service as an example. To implement the Wi-Fi service, system developers only need to define its interfaces as an AIDL description, and then implement the corresponding server-side logic (`WifiService`) and client-side wrapper (`WifiManager`) (see Fig. 2). The serialization, transmission, and de-serialization of the interface parameters are handled by the codes automatically generated from the AIDL file. Specifically, when the client invokes some RPC method in the client-side wrapper `WifiManager`, the Proxy class `IWifiManager.Stub.Proxy` will marshall the input parameters in a `Parcel` object and send it across the process boundary via the `Binder` driver. The `Binder` library at the server-side will then unmarshall the parameters and invoke the `onTransact` function in the Stub class `IWifiManager.Stub`. This eventually invokes the service logic programmed in `WifiService`. Fig. 2 provides a clear illustration of the entire process.

4. BINDER: THE ATTACK SURFACE

The `Binder` driver serves as the boundary between two communicating parties and separates them as client and server. Existing attacks on this interface typically involve directly injecting a crafted transaction via the `Binder` interface. In theory, at which layer is a transaction injected at the client-side should not affect the security of the Android system — the server-side should always be robust on its own. This is probably a best engineering practice for any system that adopts a client/server model. However, as we will show later, we found this guideline is frequently overlooked in the implementation of Android system services. Here, we review 100+ vulnerabilities found in six major Android versions and try to summarize what mistakes have the system developers made that turn the `Binder` interface into a tempting attack surface, and why system developers keep making these seemingly simple mistakes? All the vulnerabilities discussed in this section are discovered by *BinderCracker*, the first context-aware fuzzing framework for Android IPCs. We will detail the design of *BinderCracker* in the next section.

4.1 Attack Model

In this paper, we assume the adversary is a malicious app developer trying to sabotage the robustness or the integrity of Android system services. A system service can be generic, existing in Android framework base, or vendor-specific, introduced by device manufacturers. The attacker launch attacks by directly injecting crafted transactions into the `Binder` interface. The resulted consequences can be from Denial-of-Service (DoS) attack to privileged code execution, depending on the payload and the target of the malicious transaction. We assume the attacker has no root permission and cannot penetrate the security of OS kernel. Fig. 3 illustrates a typical attack scenario.

4.2 Overview of Vulnerabilities

We identified 137 vulnerabilities on the `Binder` attack surface by testing 6 major versions of Android: 4.1 (JellyBean), 4.2 (JellyBean), 4.4 (KitKat), 5.0 (Lollipop), 5.1 (Lollipop) and 6.0 (Marshmallow). Note that the number of discovered vulnerabilities have already been de-duplicated across versions. Specifically, we examined more than 98 generic system services (by Google) and 72 vendor-specific services (by Samsung), which covers more than 2400 low-level RPC methods. The majority of the vulnerabilities are in Android framework while 15 of them are in vendor-specific (Samsung) services. All our experiments are conducted by running *BinderCracker* on official firmwares from major device manufacturers (see Fig. 4). An official firmware went through extensive testing by the vendors and is believed to be ready for a public release. Each firmware is tested in the initial state, right after it is installed. We didn't install any third-party app or change any configuration except for turning on the adb debugging option, ruling out the influence of external factors.

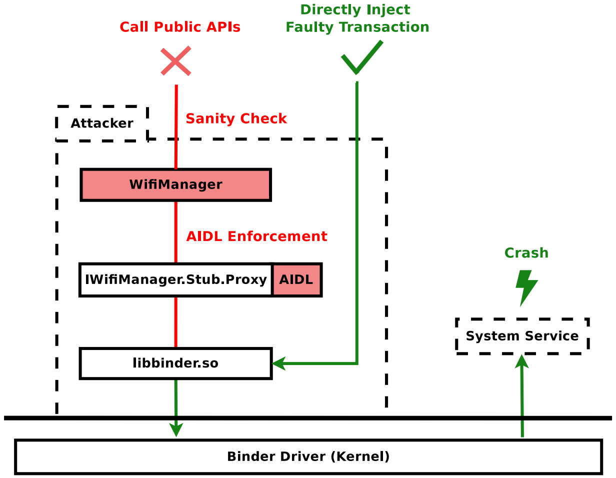


Figure 3: A typical attack scenario. By injecting faulty transactions via the `Binder` driver, an attacker can bypass the sanity check on public API and AIDL enforcement, and directly challenge the server-side.

Version	*API*	*Market*	*Device*	*Build #*
4.1.1	16	9.0%	Galaxy Note 2	JRO03C
4.2.2	17	12.2%	Galaxy S4	JDQ39
4.4.2	19	36.1%	Galaxy S4	KOT49H
5.0.1	21	16.9%	Nexus 5	LRX22C
5.1.0	22	15.7%	Nexus 5	LMY47I
6.0.0	23	0.7%	Nexus 5	MRA58K

Figure 4: List of Android ROMs we tested using *BinderCracker*.

An RPC method is found to be *vulnerable* if testing it resulted in a fatal exception, crashing part of, or the entire Android Runtime. Each unique crash (stack traces) under an RPC interface is further referred to as an *individual vulnerability*. For each vulnerability reported here, we followed the process of: 1) identify it on an official ROM, 2) manually confirm that it can be reproduced, and 3) inspect the source codes for a root cause analysis. For vendor-specific vulnerabilities of which source codes are not available, such as many of the customized system services provided by Samsung, we only record the stack trace. Fig. 5 list the number of vulnerabilities grouped by the exception types in the crash traces. The security implications of the identified vulnerabilities will be further reviewed in Section 6.

4.3 Root Cause Analysis

The direct causes of crashes are uncaught exceptions such as `NullPointerException` or `SEGV_MAPPER`, but the fundamental cause behind them is deeper. For each crashed system service of which source codes are available, we looked into the source codes and analyzed the root causes of the vulnerabilities. We noticed that sanity checks are most extensive around client-side public APIs, but are sporadic/careless after this line. This suggests that many system developers only considered the exploitation of public APIs, thus directly injecting faulty transactions to the `Binder` driver creates many scenarios that are believed to be 'unlikely' or 'impossible' in their mindset. Here, we highlight some of the new attack vectors enabled by attacking the `Binder` interface which contribute to most of the vulnerabilities we identified.

First, an attacker can manipulate RPC parameters that

Level	*Exception Type*	*Count*
Java	NullPointerException	29
	IllegalArgumentException	9
	OutOfMemoryError	8
	RuntimeException	8
	IllegalStateException	4
	StackOverflowError	4
	UnsatisfiedLinkError	2
	ArrayIndexOutOfBoundsException	2
	OutOfResourcesException	1
	SecurityException	1
	StringIndexOutOfBoundsException	1
	IOException	1
	BadParcelableException	1
Native	SEGV_MAPPER	31
	SI_TKILL	29
	BUS_ADRALN	4
	SEGV_ACCERR	1
	SI_USER	1

Figure 5: Vulnerabilities grouped by types of exception.

are not exposed via public APIs. For example, `IAudioFlinger` provides an RPC method `REGISTER_CLIENT`. This method is only implicitly called in the Android middleware and is never exposed via public interfaces. Therefore, the developers of this system service may not expect an arbitrary input from this RPC method and didn't perform a proper check of the input parameters. In our test, sending a list of null parameters via the `Binder` driver can easily crash this service. This suggests that developers should not overlook RPC interfaces that are private or hidden.

Second, an attacker can bypass sanity checks around the public API, no matter how comprehensive they are. For example, the `IBluetooth` service provides a method called `registerAppConfiguration`. All of the parameters of this RPC method are directly exposed via a public API and there are multiple layers of sanity check around this interface. Therefore, if there is an erroneous input from the public API, the client will throw an exception and crash without even sending the transaction to the server side. However, using our approach, an attack transaction is directly injected to the `Binder` driver without even going through these client-side checks. This suggests that the server should always double-check input parameters on its own.

Third, an attacker can exploit the serialization process of certain data types and create inputs that are hazardous at the server side. For example, `RemoteView` is a Parcelable object that represents a group of hierarchical views. It contains a loophole in its de-serialization module which can cause a `StackOverflow` exception. As shown in Fig. 6, a bad recursion will occur if the input Parcel object follows a certain pattern. By directly manipulating the serialized bytes of the Parcel sent via the `Binder` driver, this loophole can be triggered and crash the server. This suggests that RPC methods with serializable inputs require special attention and sanity check is also essential in the de-serializaiton process.

These common mistakes made by system developers indicate there is a misconception of where the security boundary is for Android system services — many may assume the security/trust boundary is at the client-side public APIs, and whatever happens thereafter is free from obstruction since it is already in the system zone. This mis-conception is understandable since Android provides the convenient abstraction of AIDL and automatically generate codes that serialize, send, receive, de-serialize RPC parameters. This at one side relieves the developers from implementing their own IPC stack, but at the other side hide all the details about RPC and `Binder`. In other words, even though Android system services depend on IPC extensively, the developers are likely to be agnostic of that. Therefore, we advocate the importance of introducing automatic testing and protection at the `Binder` surface — the actual security boundary.

```
android.widget.RemoteViews

private RemoteViews(Parcel parcel,
    BitmapCache bitmapCache) {

    int mode = parcel.readInt();

    ...

    if (mode == MODE_NORMAL) {
      ...
    } else {
      // recursively calls itself
      mL = new RemoteViews(parcel,
    mBitmapCache);
      // recursively calls itself
      mP = new RemoteViews(parcel,
    mBitmapCache);
      ...
    }

    ...
}
```

Figure 6: The constructor of the `RemoteView` class contains a loophole which can cause a `StackOverflow` exception. Specifically, a bad recursion will occur if the input Parcel object follows a specific pattern.

5. EFFECTIVE VULNERABILITY DISCOVERY

In this section, we explain how to conduct effective vulnerability discovery through the `Binder` interface. A naive approach is to issue transactions containing random bytes, also known as black-box fuzzing. In our experiment, we found that when using black-box fuzzing, almost all of the vulnerabilities are found within the first few fuzzing transactions, and a longer fuzzing time did not lead to the discovery of new bugs. The deficiency of black-box fuzzing is largely due to the extremely large and complex fuzzing space of this attack surface, and drives us to develope more sophisticated fuzzing techniques.

5.1 Unique Challenges

There are some unique challenges in fuzzing the `Binder` surface. First, a `Binder` transaction may contain non-primitive data types which result in complicated and hierarchical input schema. This affects 48% of all `Binder`-based RPCs and makes it difficult to fuzz according to parameter types. Moreover, the list of parameters included in a transaction is often dynamic instead of static. For example, when a transaction takes a `Bundle` object as input, what this object contains highly depends on the runtime status, and cannot be simply captured by a static interface description. All of these make it difficult to generate schemas of Binder-based transactions in an effective and reliable way.

Second, inter-dependencies often exist between Binder transactions. We found that 37% user-level RPC invocations require an input parameter that is the output of previous transactions. Note that, these input parameters are remote handlers that cannot be recorded in the form of raw bytes and can only be generated by executing the same transactions. This process is extremely crucial if we want to fuzz dynamically generated system services. While we can directly retrieve the handler of a statically cached system service and start to fuzz it, the handler of dynamically generated system services can only be retrieved by replaying the sequence of transactions it depends on.

Besides these two challenges, our design options are also restricted by the (un)availability of the source codes for the system services under test. For example, in a typical Samsung Galaxy device, there are more than 70 vendor-specific system services, the source codes of which are clearly unavailable. Even for system services that are included in the core Android framework, their source codes are typically proprietary when related to crypto or interactions with OEM hardwares. The scenario becomes more exaggerated if considering services exported by system or user-level apps.

5.2 BinderCraker: Design Overview

We present an overview of the design of *BinderCracker*. Our design addresses the above challenges by adopting a context-aware, replay-based approach which actively manages the dependencies across transactions. Specifically, *BinderCracker* includes a recording component, implemented as an Android extension (custom ROM), and a fuzzing component, implemented as a user-level app. The recording component collects detailed information of different Binder transactions and the fuzzing component tries to replay and mutate each recorded transaction for fuzzing purposes.

The recording component constructs and records the schema (parameter types and structure) of each transaction during runtime by instrumenting the (de)-serialization process of the Binder transaction. This is different from the approach that parses RPC interface (AIDL) files and does not require access to the source codes. Specifically, it monitors and records how each parameter is unmarshalled from the Parcel object. This instrumentation works recursively with the (de)-serialization functions, and thus can understand and record complicated, hierarchical schemas and non-primitive types. In additional to recording the transaction schema, *BinderCracker* automatically matches the inputs and outputs of adjacent transactions and constructs a dependency graph among them. This dependency graph captures the sequence (tree) of transactions required to generate the inputs of a target transaction (see Fig. 7). This allows *BinderCracker* to automatically manage the dependencies between transactions and reconstruct the dynamic context each transaction requires. For example, certain system services, such as IGraphicBufferProducer, are dynamically initialized instead of statically cached in the system_server. Fuzzing it typically requires manually writing a code section that first generates this service and then using reflections on private APIs to retrieve the handler, which is tedious and not scalable. *BinderCracker* greatly simplifies this process since all the transactions that generate this service will be replayed automatically before the actual fuzzing process.

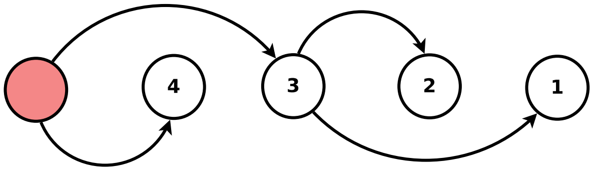


Figure 7: When fuzzing a transaction, we need to replay the supporting transactions according to their relative order in the dependency graph. This way, all the remote objects this transaction requires will be reconstructed during runtime.

```
struct binder_transaction_data {
  union {
    size_t handle; // (1).target service
    void *ptr;
  }target;
  void *cookie;
  unsigned int code; // (2).RPC method
  unsigned int flags;
  pid_t sender_pid;
  uid_t sender_euid;
  size_t data_size;
  size_t offsets_size;
  union {
        struct {

           binder_uintptr_t buffer;
           binder_uintptr_t offsets;
        } ptr;
        __u8 buf[8];
    } data; // (3).transactional data
};
```

Figure 8: The data struct sent through the Binder diver via the ioctl libc call. This struct contains three important pieces of information we need to modify to send a fuzzing transaction.

After recording the seed transactions and their dependencies, we need to utilize them to fuzz a system service. The fuzzing component of *BinderCracker* has a replay engine built-in and is implemented as a user-level app. Basically, it is manipulating (either directly or indirectly) a binder_transaction_data struct sent to the Binder driver. This data struct contains three important pieces of information we need to modify to send a fuzzing transaction and has the format as shown in Fig. 8. The target.handle field specifies the service this transaction is sent to. The code field represents a specific RPC method we want to fuzz. The data struct contains the serialized bytes of the list of parameters for the RPC method, which is inherently a Parcel object. Parcel is a container class that provides a convenient set of serialization and de-serialization methods for different data types. Both the client and the server work directly with this Parcel object to send and receive the input parameters. Later in this section, we will elaborate on how to modify the handle and code variables to redirect the transaction to a specific RPC method of a specified service, and how to fuzz the Parcel object to facilitate testing with different policies.

5.3 Transaction Redirection

There is a one-to-one mapping from the handle variable

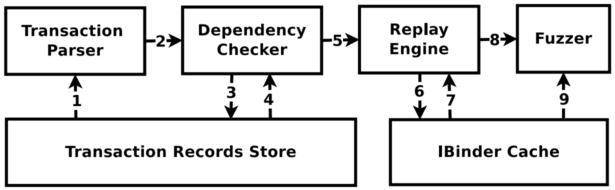


Figure 9: How does *BinderCracker* generate semi-valid fuzzing transactions from seed transactions.

in the binder_transaction_data object to system service. This mapping is created during runtime and maintained by the Binder driver. Since the client has no control over the Binder driver, it cannot get this mapping directly. For system services that are statically cached, we can get them indirectly by querying a static service manager which has a fixed handle of 0. This service manager is a centralized controller for service registry and will be started before any other services. By sending a service interface descriptor (such as android.os.IWindowManager) to the service manager, it will return an IBinder object which contains the handle for the specified service. For system services that are dynamically allocated, we can retrieve them by recursively replaying the supporting transactions that generate these services (see Fig. 7).

After getting the handle of a system service, we need to specify the code variable in the binder_transaction_data object. Each code represents a different RPC method defined in the AIDL file. This mapping can be found in the Stub files which are automatically generated from the AIDL file. The code variable typically ranges from 1 to the total number of methods declared in the AIDL file. For native system services that are not implemented in Java, this mapping is directly coded in either the source files or the header files. Therefore, we scan both the AIDL files and the native source codes of Android to construct the mapping between transaction codes and RPC methods.

5.4 Transaction Fuzzing

After being able to redirect a Binder transaction to a chosen RPC method of a chosen system service, the next step is to manipulate the transaction data and create faulty transactions that are unlikely to occur in normal circumstances. Here, *BinderCracker* utilize our context-aware replay engine to generate semi-valid fuzzing transactions. A transaction is said to be *semi-valid* if all of the parameters it contains are valid except for one. Semi-valid transactions can dive deeper into the program structure without being early rejected, thus is able to reveal more in-depth vulnerabilities.

In summary, *BinderCracker* maintains both the type hierarchy and dependency graph when recording a seed transaction. These information capture the semantic and context of each transaction and help *BinderCracker* generate semi-valid fuzzing transactions. Specifically, it follows the process illustrated in Fig. 9. For each seed transaction we want to fuzz, we first parse the raw bytes of the transaction and unmarshall non-primitive data types into an array of primitive types (step 1). This utilizes the type hierarchy recorded with the seed transaction. Then, we check the dependency of the transaction (step 2) and retrieve all the supporting transactions (steps 3, 4). This step utilizes the dependency graph recorded with the seed transaction. After that, we need to replay the supporting transactions (step 5) to generate and cache the remote IBinder object handles (steps 6, 7). Finally, the fuzzer can start to generate semi-valid fuzzing transactions by mutating each parameter in the seed transaction according to their data types (steps 8, 9). For example, for numerical types such as Integer, we may add or substrate a small delta from the current value or change it to Integer.MAX, 0 or Integer.MIN; for literal types such as String, we may randomly mutate the bytes contained in the String, append new content at start or end, or insert special characters at certain locations.

After sending a faulty transaction to a remote service, there are a few possible responses from the server-side. First, the server detects the input is invalid and rejects the transaction, writing an IllegalArgumentException message back to the client. Second, the server accepts the argument and starts the transaction, but encounters unexpected states or behaviors and catches some type of RuntimeException. Third, the server doesn't catch some bizarre scenarios, causes a Fatal Exception and crashes itself. In this paper, we focus on the last type of responses, as it is most critical and has disastrous consequences.

5.5 Experimental Results

We collected more than one million valid seed transactions by running 30 popular apps in two latest Android versions (Android 5.1 and Android 6.0). For each RPC interface, we sampled the transactions and selected those with unique transaction schema/structures. Based on this seed dataset, we performed fuzzing test on more than 445 RPC methods of 78 system services. In total, we identified 89 vulnerabilities in Android 5.1 and Android 6.0 which is 7x more than simple fuzzing with the same time spent. Compared to the vulnerabilities identified using simple black-box fuzzing, the vulnerabilities exposed by context-aware fuzzing are more interesting and have severer security implications — we start to identify buffer overflow, serialization bugs in deeper layers of the code, instead of just simple crashes or parsing errors (see Section 6 for details).

Moreover, since the approach *BinderCracker* adopts is generic, we can easily configure it to fuzz higher-level abstractions or protocols. For example, Intent is a high-level abstraction built on top of the Binder RPCs. It is used as a user-level communication primitive to launch apps, services or trigger broadcast receivers. With some simple configuration, we can turn *BinderCracker* into an Intent fuzzer. Specifically, we configure *BinderCracker* to only fuzz three RPC interfaces that the Intent communication mechanism is built upon, and only mutate the Intent parameter in these RPC calls. This makes *BinderCracker* a useful Intent fuzzer that automatically tracks and utilizes the internal type hierarchy of Intent extras (Bundle). Fig. 10 illustrates the input structure of an example Intent we fuzz. In total, *BinderCracker* identified more than 20 vulnerabilities in the Intent communication, many of which exist in the de-serialization process of Intent.

6. DEFENSES

New vulnerabilities are still emerging on the Binder attack surface whenever there is a major upgrade of the Android code base. This is because, considering the code size of Android, it is almost impossible to prevent the developers from writing buggy codes. We discussed how to conduct effective precautionary testing to help expose vulnerabilities

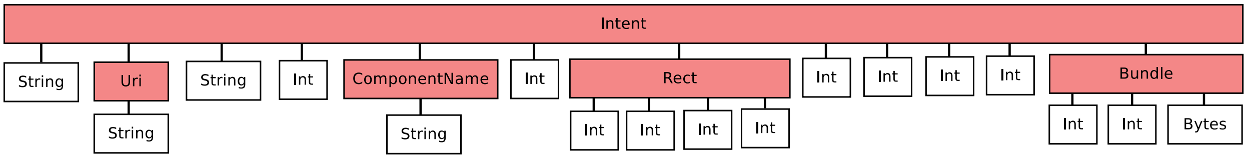


Figure 10: The internal type structure of a non-primitive data type, `Intent`, generated by recording the de-serialization process of each non-primitive type. Note that this type structure is dynamic — it depends on what has been put into this `Intent` during runtime.

before releasing the new ROM, and also explained why it is very difficult, if not impossible, to conduct runtime defense in existing Android systems. The major obstacle in developing runtime defenses is the lack of transparency/auditing on IPC transactions. When a system service fails, no one knows why it crashed and who caused its crash without proper OS-level support. Thus, a system-level IPC diagnosis tool is essential for any in-depth runtime analysis, such as attack attribution and cross-device analytics.

6.1 Precautionary Testing

Before releasing a new ROM, developers can conduct precautionary testing. The defense can be done early, in the development phase of each system service, or later, after the entire ROM gets built.

Android has already adopted a static code analysis tool, `lint`, to check potential bugs and optimizations for correctness, security, performance, usability, accessibility and internationalization [20]. Specifically, `lint` provides a feature that supports inspection with annotations. This allows the developer to add metadata tags to variables, parameters and return values. For example, the developer can mark an input parameter as `@NonNull`, indicating that it cannot be `Null`, or mark it as `@IntRange(from=0,to=255)`, enforcing that it can only be within a given range. Then, `lint` automatically analyzes the source codes and prompts potential violations. This can be extended to support inspections of RPC interfaces, allowing developers to explicitly declare the constraints for each RPC input parameter. This way, many potential bugs can be eliminated during the development phase. This defense is practical and comprehensive but requires system developers to specify the metadata tags for each RPC interface.

We can also conduct precautionary testing during runtime after the ROM has been built. Our system, *BinderCracker*, is effective in identifying vulnerabilities and can be used as an automatic testing tool. By fuzzing various system services with different policies, a large number of vulnerabilities can be eliminated before reaching the end-users. Actually, many severe vulnerabilities [6–9, 29] could have been avoided if a tool like *BinderCracker* had been deployed. Note that the effectiveness of *BinderCracker* depends on the quality and coverage of the seed transactions. Besides collecting execution traces of a large number of apps, another potential way of generating a comprehensive seed dataset is to incorporate the functional unit tests of each system service.

6.2 Security Implications

Most of the vulnerabilities *BinderCracker* discovered (more than 90%) can be used to launch a Denial-Of-Service (DoS) attack. Some of them are found to be able to crash the entire Android Runtime, while others can cause specific system services or system apps to fail. Fig. 11 shows the distribution of the affected services (apps). When launching a DoS attack, the attacker can trigger a crash either consistently or only under certain conditions, for example, when a competitor's app is running. This can create the impression that the competitor's app is buggy and unusable. We even identified multiple vulnerabilities (in the de-serialization process of `Intent`) that can cause targeted crash of almost any system/user-level apps, without crashing the entire system. Specifically, an attacker can craft an Intent that contains a mal-formated `Bundle` object and send to the target app. This will cause a crash during the de-serialization process of the `Intent` object before the target app can conduct any sanity check. Moreover, it can be very challenging to identify the attacker app under these scenarios because the OS only knows which service/app is broken, but cannot tell who crashed it. We will discuss more about the attack attribution process in later sections.

```
native_handle_t* native_handle_create(int
    numFds, int numInts)
{
    // numFds & numInts are not checked!
    native_handle_t* h = malloc( ...
        + sizeof(int)*(numFds+numInts));

    h->version = sizeof(native_handle_t);
    h->numFds = numFds;
    h->numInts = numInts;

    return h;
}
```

Figure 12: The constructor of the `native_handle` has an Integer Overflow vulnerability that can cause a heap corruption on the server-side. This can lead to privileged code execution in `system_server`.

We also discovered a few vulnerabilities that can cause other serious security problems. We found that in several RPC methods, the server-side fails to check potential Integer overflows. This may lead to disastrous consequences when exploited by an experienced attacker. For example, in `IGraphicBufferProducer` an Integer overflow exists such that when a new `NativeHandle` is created, the server will malloc smaller memory than it actually requested (see Fig. 12). Subsequent writes to this data struct will corrupt the heap on the server-side. This vulnerability has been demonstrated to be able to achieve privileged code execution, and insert any arbitrary code into `system_server` [7]. We also found a vulnerability in `IContentService` that can lead to an infinite bootloop, which can only be resolved by factory recovery or flashing a new ROM. This is also classified as High Risk according to the official specification of Android severity levels [32].

Besides RPC methods that are not well implemented, we

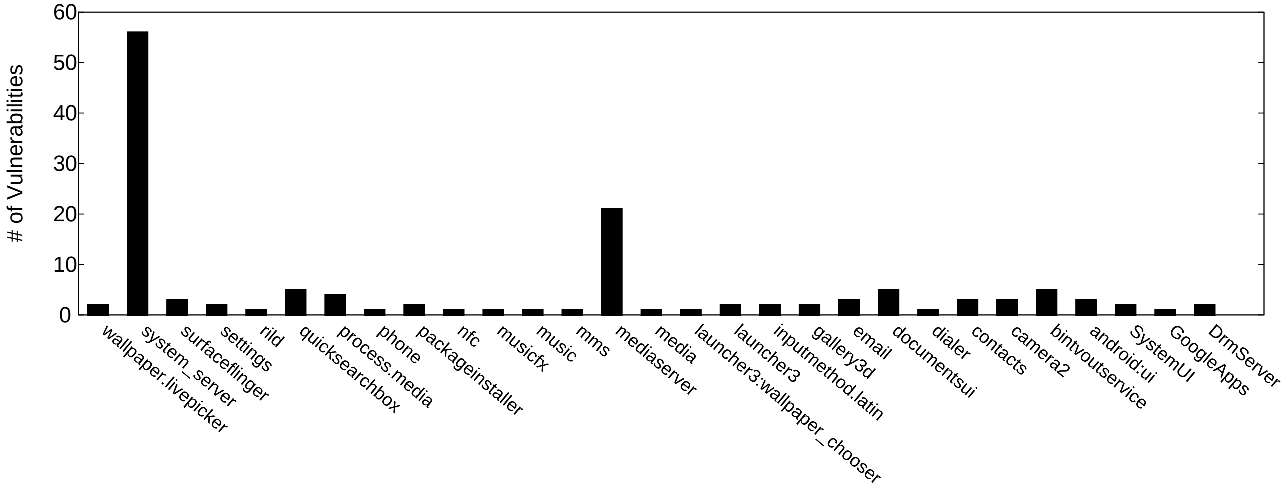


Figure 11: Many of the vulnerabilities we identified are found to be able to crash the entire Android Runtime (system_server), while others can cause specific system services (mediaserver) or system apps (nfc, contacts, etc) to fail.

also discovered RPC methods that are not properly protected by existing Permission models. In official ROMs of Samsung Galaxy 4 (Android 4.2.2 and Android 4.4.2), an attacker can reboot the device by directly sending a transaction to `PackageManagerService` via the `Binder` driver without requiring the REBOOT permission. This is critical since REBOOT is a sensitive permission only granted to system apps. The other service is `ICoverManager`, a customized service from Samsung. An attacker can invoke a certain RPC method of `ICoverManager` and block the entire screen with a pop-up blank Activity. The blank Activity cannot be revoked using any virtual or physical button and the only exit is restarting the device.

6.3 Vulnerabilities: Fixed and Unfixed

We examined how many of the vulnerabilities discovered by *BinderCracker* remain unfixed and are potentially zero-day when they are found. Our analysis is based on the public changes of the source codes across different Android versions and revisions. We skipped the 15 vulnerabilities in vendor-specific system services and 7 in generic system services due to the unavailability of source codes. Note that not all generic system services are open source, especially when it is related to decryption/encryption or interactions with OEM hardware.

Of the 115 analyzed vulnerabilities in Android code bases, only 18 have been fixed by adding additional sanity checks of input parameters. Another 12 vulnerabilities 'disappeared' during several major Android version upgrades either because 1) the corresponding source codes (or API) have been deleted; or 2) new updates in other parts of the source codes accidentally bypass the vulnerable source codes. For example, some crashes are caused by a recursive call in the `RemoteView` class (see Fig. 6). Similar crashes disappeared after Android 5.0. We looked into the source codes and found this is not because the bug has been fixed, but because in new versions of Android a faulty transaction will create an additional Exception before it reaches the vulnerable codes. The additional Exception is properly caught and accidentally avoids the fatal crash caused by the real vulnerability. We do not consider this as a 'fix' since an attacker can still recreate the crash by manually crafting a transaction which bypasses the new code updates. Fig. 13 illustrates the proportion of vulnerabilities that are fixed, disappeared and unfixed. We have already submitted all unfixed vulnerabilities to AOSP.

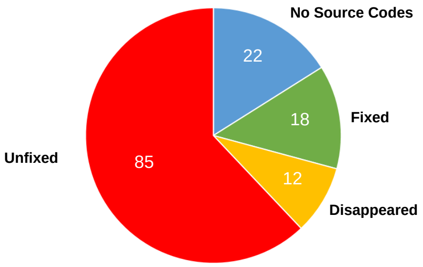
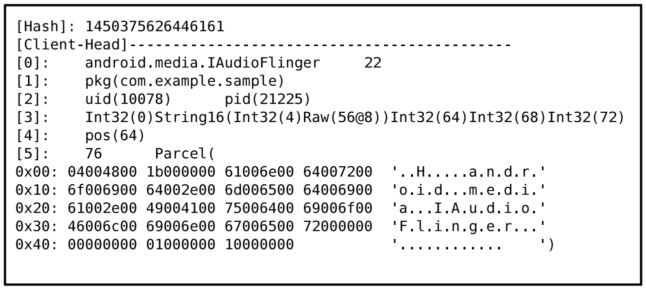


Figure 13: Number of the vulnerabilities that are fixed, disappeared and unfixed.

6.4 Runtime Diagnostics and Defenses

It will be helpful if Android can provide some real-time defense against potential vulnerabilities even after the ROM has been deployed on end-users' devices. Here, we focus on specific defenses on the `Binder` layer, excluding generic defenses such as Address Space Layout Randomization (ASLR), SELinux, etc. They have been discussed extensively elsewhere [22, 33, 35] and are not specific to our scenario. Basically, there are two potential defenses one can provide on the `Binder` surface during runtime: (i) *intrusion detection/prevention*, identifying and rejecting transactions that are malicious, and (ii) *intrusion diagnostics*, making an attack visible after the transaction has already caused some damage. Unfortunately, both approaches are not applicable in existing Android systems. Next, we explain why the first approach is inherently challenging and then describe our efforts to enable the latter.

To provide runtime intrusion prevention, one needs to perform some type of abnormality detection on incoming transactions. This works by examining the input parameters of valid/invalid RPC invocations and characterizing the rules or boundaries. However, in our case, it is not prac-



```
[Hash]: 1450375626446161
[Client-Head]--------------------------------------------
[0]:    android.media.IAudioFlinger     22
[1]:    pkg(com.example.sample)
[2]:    uid(10078)      pid(21225)
[3]:    Int32(0)String16(Int32(4)Raw(56@8))Int32(64)Int32(68)Int32(72)
[4]:    pos(64)
[5]:    76      Parcel(
0x00: 04004800 1b000000 61006e00 64007200  '..H.....a.n.d.r.'
0x10: 6f006900 64002e00 6d006500 64006900  'o.i.d...m.e.d.i.'
0x20: 61002e00 49004100 75006400 69006f00  'a...I.A.u.d.i.o.'
0x30: 46006c00 69006e00 67006500 72000000  'F.l.i.n.g.e.r...'
0x40: 00000000 01000000 10000000           '............    ')
```

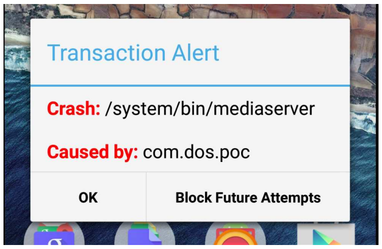
Figure 14: An example report generated by our diagnostic tools. This includes the system service under attack (0), transaction sender (1 and 2), schema (3), position of the parsing cursor (4) and raw content (5).

tical for the following reasons. First, Binder transactions occur at a very high frequency but a mobile device has only limited energy and computation power. Second, parameters in Binder transactions are very diverse, codependent, and evolving dynamically during runtime, and hence clear boundaries or rules may not exist. Third, end-users are not likely to accept even the smallest false-positive rate. One can, of course, build a very conservative blacklist-based system and hard-coding rules of each potential vulnerability in the database. However, this seems unnecessary, especially when Android nowadays supports directly pushing security updates (patches) to devices of end-users.

An alternative solution is to diagnose, instead of prevent problems. It would be helpful if we can provide more visibility on how malicious transactions actually undermine a device. Even though this cannot stop the single device from being attacked, we can still utilize the collected statistics to develop in-time security patches, benefiting the vast majority of end-users. However, it is impossible to conduct informative diagnostic on the Binder layer even for developers with adb access. This is due to a lack of transparency/auditing on IPC transactions. Essentially, when a system service fails, no one knows why it crashed and who crashed it without proper OS-level support. This, also known as *attack attribution*, has not yet received enough attention from the research community. To fill this gap, we propose a system-layer diagnostic tool and demonstrate its use for more in-depth analyses.

Our diagnostic tool provides three important functionalities: 1) when a service fails when processing an incoming transaction, the sender of the transaction will be recorded and the user will be warned with a visual prompt; 2) detailed information of the failed transaction, including the content, schema and parsing status will be dumped into a report for future forensics; 3) a signature of the transaction will be generated and the user can review it and choose to block future occurrences of the same transaction. These, together, capture a snapshot of the IPC stack in case of a potential attack. This snapshot can be triggered by a crash, for DoS attacks, or by access to sensitive/privileged APIs, for privilege escalation attacks.

The diagnostic tool is implemented in a similar way as the recording component of *BinderCracker*, by instrumenting the Binder framework and the Parcel class. We further retrieve the sender of each transaction by calling Binder.getCallingUid() in the victim system service, and get the package name of the sender by querying the PackageManagerService with the retrieved uid. The same flow is also used to support permission checks in Android system services. The schema of each transaction is constructed and maintained during runtime by tracking the de-serialization process of the Parcel. Whenever an Exception is thrown and not caught by any of the Exception handling blocks, we will dump the transaction information we maintained as a separate report. The parsing information will be appended, indicating which parameters the service is parsing (have just parsed) when the failure/attack happens. The signature of the transaction will be recorded and the user will be prompted with a Notification (see Fig. 15). End-users can choose to block transactions with the same signature in the future to mitigate continuous DoS attacks. Fig. 14 presents an example report generated by our diagnostic tool. This marks an essential step towards more sophisticated runtime analysis, such cross-device analytics.



Transaction Alert

Crash: /system/bin/mediaserver

Caused by: com.dos.poc

Figure 15: User receives a visual prompt in case of transaction failure and can choose to block it to counter continuous DoS attack.

7. DISCUSSION

We have assessed a risky attack surface comprehensively which has long been overlooked by the system developers of Android. As our experimental results demonstrated, new vulnerabilities are still emerging on this attack surface and *BinderCracker* can help eliminate potential vulnerabilities in future releases of Android. The lessons learned can transcend to other platforms facing similar issues, such as vehicular systems (CAN buses and ECUs), wearable devices, etc. We highlight that, although many systems adopt a client–server model in the design of their internal system components, they rarely follow the security standards of a real client–server model as in a networked environment. In many scenarios, a component may fall into the wrong hands and create serious security threats.

Our context-aware fuzzing is generic and not limited to system services. In fact, it also works for services exported by user-level apps. For example, Facebook alone exports more than 30 services to other apps which forms a large attack surface. By performing fuzzing on this interface, more app-level vulnerabilities are expected to be unearthed. However, due to the unavailability of source codes, it is difficult to analyze the root causes and security implications of the identified vulnerabilities. Note that, although lack of source codes won't affect the discovery of vulnerabilities, it does make it more difficult to understand their implications.

8. CONCLUSION

In this paper, we conducted an in-depth analysis on an emerging attack surface in Android. We summarized the common mistakes made by system developers that produces this attack surface and highlight the importance of testing and protection on the `Binder` interface, the actual trust boundary. We designed and implemented *BinderCracker*, a precautionary testing framework that achieves automatic vulnerability discovery. It supports context-aware fuzzing and performs 7x more effectively than a simple black-box fuzzing approach. We also addressed the urgent problem of attack attribution for IPC-based attacks, by providing OS-level runtime diagnostics support. These mechanisms are useful, practical and can be easily integrated into the development/deployment cycle of Android.

9. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261. IEEE, 2011.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[3] Android interface definition language (aidl). http://developer.android.com/guide/components/aidl.html.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[6] Cve-2015-1474. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1474.

[7] Cve-2015-1528. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1528.

[8] Cve-2015-6612. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6612.

[9] Cve-2015-6620. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6620.

[10] 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.

[11] W. Enck, P. Gilbert, 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. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[14] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[15] Fuzzing android system services by binder call. https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf.

[16] Google says there are now 1.4 billion active android devices worldwide. http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide.

[17] Hey your parcel looks bad. https://www.blackhat.com/docs/asia-16/materials/asia-16-He-Hey-Your-Parcel-Looks-Bad-Fuzzing-And-Exploiting-Parcelization-Vulnerabilities-In-Android.pdf.

[18] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[19] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83. ACM, 2011.

[20] Improving your code with lint. http://developer.android.com/tools/debugging/improving-w-lint.html.

[21] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 69–80, New York, NY, USA, 2012. ACM.

[22] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 424–439, Washington, DC, USA, 2014. IEEE Computer Society.

[23] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[24] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington,

DC, USA, 2012. IEEE Computer Society.

[25] C. Marforio, A. Francillon, S. Capkun, S. Capkun, and S. Capkun. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems*. Department of Computer Science, ETH Zurich, 2011.

[26] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[27] 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*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[28] Number of apps available in leading app stores as of july 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[29] O. Peles and R. Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[30] M. Rangwala, P. Zhang, X. Zou, and F. Li. A taxonomy of privilege escalation attacks in android applications. *Int. J. Secur. Netw.*, 9(1):40–55, Feb. 2014.

[31] R. Sasnauskas and J. Regehr. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. ACM.

[32] Security updates and resources. https://source.android.com/security/overview/updates-resources.html.

[33] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, (3):36–44, 2009.

[34] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, Mar. 2010.

[35] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[36] Smartphone os market share, q4 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[37] Accusations fly between uber and lyft. http://bits.blogs.nytimes.com/2014/08/12/accusations-fly-between-uber-and-lyft/.

[38] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.