FANS: Fuzzing Android Native System Services via Automated Interface Analysis

Baozheng Liu[1,2]*, Chao Zhang[1,2]✉, Guang Gong[3],
Yishun Zeng[1,2], Haifeng Ruan[4], Jianwei Zhuge[1,2]✉

[1]*Institute of Network Science and Cyberspace, Tsinghua University* ✉chaoz@tsinghua.edu.cn
[2]*Beijing National Research Center for Information Science and Technology* ✉zhugejw@tsinghua.edu.cn
[3]*Alpha Lab, 360 Internet Security Center* [4]*Department of Computer Science and Technology, Tsinghua University*

Abstract

Android native system services provide essential supports and fundamental functionalities for user apps. Finding vulnerabilities in them is crucial for Android security. Fuzzing is one of the most popular vulnerability discovery solutions, yet faces several challenges when applied to Android native system services. First, such services are invoked via a special inter-process communication (IPC) mechanism, namely `binder`, via service-specific interfaces. Thus, the fuzzer has to recognize all interfaces and generate interface-specific test cases automatically. Second, effective test cases should satisfy the interface model of each interface. Third, the test cases should also satisfy the semantic requirements, including variable dependencies and interface dependencies.

In this paper, we propose an automated generation-based fuzzing solution FANS to find vulnerabilities in Android native system services. It first collects all interfaces in target services and uncovers deep nested multi-level interfaces to test. Then, it automatically extracts interface models, including feasible transaction code, variable names and types in the transaction data, from the abstract syntax tree (AST) of target interfaces. Further, it infers variable dependencies in transactions via the variable name and type knowledge, and infers interface dependencies via the generation and use relationship. Finally, it employs the interface models and dependency knowledge to generate sequences of transactions, which have valid formats and semantics, to test interfaces of target services. We implemented a prototype of FANS from scratch and evaluated it on six smartphones equipped with a recent version of Android, i.e., android-9.0.0_r46 , and found 30 unique vulnerabilities deduplicated from thousands of crashes, of which 20 have been confirmed by Google. Surprisingly, we also discovered 138 unique Java exceptions during fuzzing.

*Part of this work was done during Baozheng Liu's research internship at Alpha Lab of 360.

1 Introduction

Android has become the most popular mobile operating system, taking over 85% markets according to International Data Corporation[1]. The most fundamental functions of Android are provided by Android system services, e.g., the camera service. Until October 2019, hundreds of vulnerabilities related to Android system services had been reported to Google, revealing that Android system services are still vulnerable and attractive for attackers. A large portion of these vulnerabilities reside in native system services, i.e., those mainly written in C++. Vulnerabilities in Android native system services could allow remote attackers to compromise the Android system, e.g., performing privilege escalation, by means of launching IPC requests with crafted inputs from third-party applications. Finding vulnerabilities in Android native system services is thus crucial for Android security.

However, to the best of our knowledge, existing researches paid little attention to Android native system services. Apart from a non-scalable manual approach [7], two automated fuzzing solutions have been proposed to discover vulnerabilities in Android system services. The first one is BinderCracker [6], which captures input models of target services by recording requests made by 30 popular applications. An inherent disadvantage of this approach is that it cannot recover precise input semantics, e.g., variable names and types. Also, it will miss rarely-used or deeply-nested interfaces, due to the incomplete testing. The other one is Chizpurfle [10], which utilizes Java reflection to acquire parameter types of interfaces to test vendor-implemented Java services. However, such a method cannot be used to retrieve the input model of Android native system services.

In Android, system services are registered to the `Service Manager`. User apps query the manager to get the target service's interface (encapsulated in a proxy Binder object), then invoke different transactions provided by this interface via a unified remote procedure call (RPC) interface named `IBinder::transact(code,data,reply,flags)`, where,

[1]https://www.idc.com/promo/smartphone-market-share/os

(1) `code` determines the target transaction to invoke, and (2) inputs of the transaction are marshalled into the serialized `parcel` object `data`. Thus, we could utilize this unified IPC method to test all system services. To thoroughly test target services, we could first find all interfaces and available transactions, and then invoke them with input `data` satisfying service-specific formats and semantic requirements. Specifically, there are three challenges to address:

C1: Multi-Level Interface Recognition. In addition to the (top-level) interfaces registered in the `Service Manager`, there are nested multi-level interfaces, which could be retrieved via the top-level interface and invoked by user apps. For example, the `IMeoryHeap` interface is buried at the fifth-level (i.e., invoked via four layers of interfaces). Therefore, we need to recognize all top-level interfaces and nested multi-layer interfaces, in order to systematically test Android system services. Given that many interfaces are defined in Android Interface Definition Language (AIDL) rather than C++ and dynamically generated during compilation, we have to take them into consideration as well.

C2: Interface Model Extraction. For each interface, we need to get the list of supported transactions (i.e., `code`) to test, and then provide input `data` to invoke each transaction. To improve the fuzzing effectiveness, the input `data` should follow grammatical requirements of target interfaces. Manually providing the grammar knowledge is not scalable. Automatically extracting such knowledge from the large volume of Android source code is also challenging. First, the grammar is specific to an individual transaction, and thus we have to recognize all available transactions and extract grammars for each of them. Second, the grammar requirements co-exist with the path constraints, e.g., branch conditions, loop conditions and even nested loops, making it hard to be extracted and represented.

C3: Semantically-correct Input Generation. Android itself performs many sanity checks (e.g., size check) on the input `data`. Therefore, inputs that do not meet semantic requirements can hardly explore deep states or trigger vulnerabilities. There are many types of semantic requirements, including variable names and types, and even dependencies between variables or interfaces. For instance, a variable named `packageName` indicates an existing package's name is required; a variable of an enumeration type can only have a limited set of candidate values; a variable in current transaction may depend on another variable in either the current or previous transaction, and even an interface may depend on another interface. Recognizing such semantic requirements and generating inputs accordingly are important but challenging.

Our Approach. In this paper, we propose a generation-based fuzzing solution FANS to address the aforementioned challenges. To address the challenge *C1*, FANS first recognizes all top-level interfaces by scanning service registration operations, and utilizes the fact that deep interfaces are generated by invoking the special method `writeStrongBinder` to identify multi-level interfaces. For *C2*, we notice that, Android system services always use a set of specific deserialization methods (e.g., `readInt32`) to parse input data. By recognizing the invocation sequence of such methods, we could infer the grammar of a valid input. To preserve the knowledge of variables' names and types, we choose to extract the deserialization sequence (i.e., the input grammar) from abstract syntax tree (AST). For *C3*, we will utilize the variable name and type knowledge extracted from the AST to generate proper inputs and recognize intra-transaction variable dependency. Further, we rely on the fact that a dependent transaction will deserialize data serialized by the depended transaction, to recognize inter-transaction variable dependency. Moreover, we rely on the generation and use relationship between interfaces to infer their dependencies.

We implemented a prototype of FANS from scratch, intermittently examined it on six mobile phones equipped with the recent Android version android-9.0.0_r46 for about 30 days. FANS has discovered 30 unique vulnerabilities deduplicated from thousands of crashes. To our surprise, FANS also found 138 unique Java exceptions, yielded by Java applications that might depend on Android native system services. Besides, we dig into the code and observe that some Android native system services would also invoke Java methods. We have submitted all native bugs to Google, and received 20 confirmations. As for the Java exceptions, we are working on examining them manually and submitting them to Google. To facilitate future research, we open source the prototype of FANS at https://github.com/iromise/fans.

Contributions. In summary, this paper makes the following contributions:

- We systematically investigated the dependency between interfaces in Android native system services, and unearthed deeper multi-level interfaces.
- We proposed a solution to automatically extract input interface model and semantics from AST. This method can be applied to other interface-based programs.
- We proposed a solution to infer inter-transaction dependencies, by utilizing variable name and type knowledge in serialization and deserialization pairs in different transactions.
- We implemented a prototype of FANS to systematically fuzz Android native system services, and have found 30 unique native vulnerabilities and 138 unique Java exceptions.

2 Background

In this section, we start by introducing the Android system service. Then we provide the research scope of this paper.

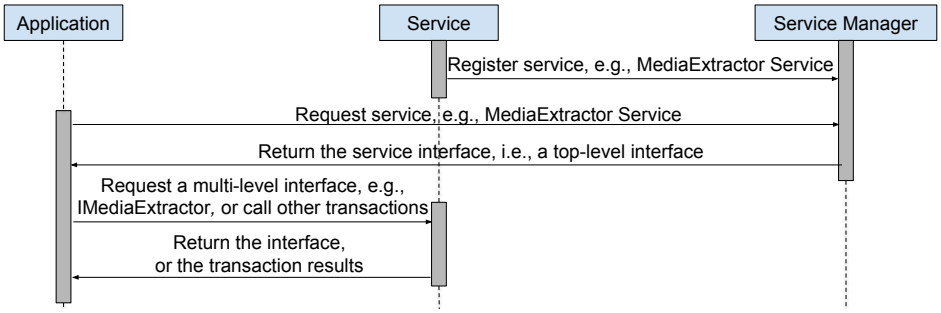

Figure 1: Application-Service Communication Model

2.1 Android System Services

System services are essential parts of Android, providing the most fundamental and core functionalities.

Systematization of Android System Services. Depending on the programming language, Android system services can be divided into two categories: (1) *Java system services*, which are implemented mainly using Java, e.g., activity manager. (2) *native system services*, which are implemented mainly using C++, e.g., camera service. Some Android native system services run as daemons, e.g., *netd*. Note that a native service might sometimes call java code and vice versa.

From another perspective, the services are divided into three domains since Android 8, including normal domain, vendor domain and hardware domain. Services in normal domain are services directly located in Android Open Source Project (AOSP), while services inside vendor domain and hardware domain are related to vendors and hardware respectively.

Application-Service Communication Model. Figure 1 illustrates the workflow of the *application-service communication* in Android. A service will first register itself into the service manager, and then listen to and handle requests from applications. On the other hand, an application will query the service manager to obtain the interface (encapsulated in a proxy Binder object) of the target service, which is denoted as a *top-level interface*. Then, it can utilize the top-level interface to retrieve a multi-level interface or to call transactions provided by the interface to perform certain actions. Further, the application could retrieve deeper multi-level interfaces and invoke corresponding transactions. Apart from the entities illustrated in the figure, there is another important entity, i.e., *binder driver*, which bridges the communication between applications and services. However, as the binder driver is not strictly relevant to our research, we omit it in the figure.

Interfaces in Android System Services. As mentioned earlier, apps invoke target transactions in top-level interfaces via a unified RPC interface `IBinder::transact(code, data,reply,flags)`. Therefore, it implies that on the service side there is a dispatcher responsible for handling the request based on the transaction `code`. This dispatcher is defined in a unified method `onTransact(code, data, reply, flags)`. This dispatcher (or the target transaction) will then deserialize the input `data` and perform the action requested by the client. In general, every service has a set of methods that can be called through RPC. They are declared in a base class, but implemented in the client-side proxy and the server-side stub separately. The binder driver bridges the proxy and stub objects to communicate.

This mechanism also applies to multi-level interfaces, as multi-level interfaces share the same architecture with top-level interfaces. However, unlike top-level interfaces, the Binder objects corresponding to multi-level interfaces are not available in the *service manager*, and could only be retrieved via top-level interfaces.

Besides, not all interfaces are statically defined in C++, and some of them are defined in the Android Interface Definition Language (AIDL). When building an Android image, AIDL tools will be invoked to dynamically generate proper C++ code for further compilation.

2.2 Research Scope

In this paper, we focus on discovering vulnerabilities in the Android *native* system services, which are registered in the *service manager* and belong to the normal domain. To the best of our knowledge, existing researches have paid little attention to them. Meanwhile, as all Android system services share the same architecture in the aspect of communication and interface implementation, the scheme proposed in this paper can be applied to other types of services as well.

3 Design

To find vulnerabilities in Android native system services, we propose a generation-based fuzzing solution FANS, and present its design in this section.

3.1 Design Choices

RPC-centric testing: There are several alternative solutions to testing Android native system services. A straightforward solution is to test target transactions by directly injecting

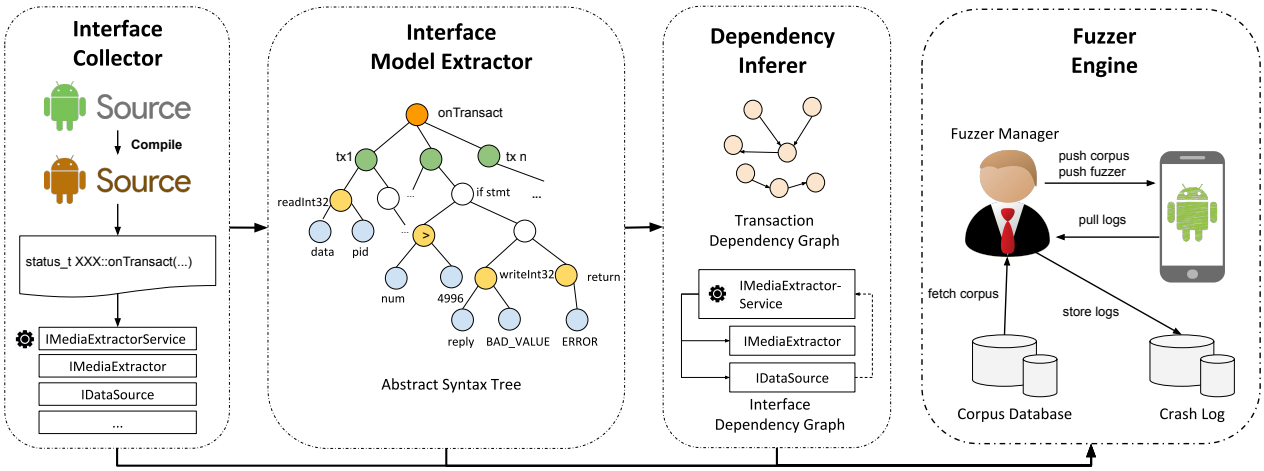

Figure 2: Overview of FANS.

service-specific events to the system, without calling the unified binder communication interface `transact`. However, there are a lot of engineering challenges to address in order to inject events to different services located in different processes. More importantly, vulnerabilities found in this way are likely to be false alarms, because the adversary in practice cannot generate arbitrary events. Instead, the adversary has to interact with target services via the IPC interface, and could only produce a limited number of events for the following two reasons: (1) the binder IPC mechanism will perform some sanity checks, e.g., on packet size; and (2) the data marshalled into a parcel might depend on some dynamic system states and are thus not arbitrary. To reduce false positives, we choose to test target services via the RPC interface, as could be done by an adversary.

Generation-based fuzzing: In general, there are two types of fuzzers: mutation-based [4, 25], which generates new test cases by mutating existing test cases, and generation-based [5, 19][2], which generate test cases according to an input specification. Mutation-based fuzzers are likely to generate test cases of invalid formats or semantics, which cannot be correctly deserialized or processed by target services. Therefore, such fuzzers tend to have low code coverage of target services and may miss many potential vulnerabilities. To reduce false negatives, we choose to test target services with generation-based fuzzing.

Learn input model from code: Generation-based fuzzers rely on input model knowledge to generate valid and effective test cases. A large number of generation-based fuzzers, including PEACH [5], Skyfire [20] and Syzkaller [19], rely on grammar files produced by human to generate test cases, which generally require huge manual efforts and are currently unavailable for Android services. Another line of works, e.g., BinderCracker [6], learn from existing transactions to generate new inputs. This type of solutions is in general incomplete, since it relies on the completeness of example transactions and will probably overlook rarely-used transactions. Moreover, the input model learned in this way is in general inaccurate, since only transaction data is given. On the other hand, we notice that the input model knowledge is buried in the source code, and choose to analyze Android source code to automatically retrieve the input model.

[2]Some generation-based fuzzers also utilize mutations to increase the diversity of test cases.

3.2 Overview

Figure 2 illustrates the design overview of our solution FANS. First, the *interface collector* (Section 3.3) collects all interfaces in target services, including top-level interfaces and multi-level interfaces. Then *interface model extractor* (Section 3.4) extracts input and output formats as well as variable semantics, i.e., variable names and types, for each candidate transaction in each collected interface. The extractor also collects definitions of structures, enumerations and type aliases that are relevant to variables. Next, the *dependency inferer* (Section 3.5) infers interface dependencies, as well as intra-transaction and inter-transaction variable dependencies. Finally, based on the above information, the *fuzzer engine* (Section 3.6) randomly generates transactions and invokes corresponding interfaces to fuzz native system services. The fuzzer engine also has a *manager* responsible for synchronizing data between the host and the mobile phone being tested.

3.3 Interface Collector

As demonstrated in Section 2.1, top-level or multi-level interfaces both have the `onTransact` method to dispatch transactions. Thus, we could utilize this feature to recognize interfaces. We do not directly scan C/C++ files in the AOSP codebase for the `onTransact` method, though. Instead, we examine every C/C++ file that appears as a source in AOSP compilation commands, so that we can collect interfaces that are dynamically generated by AIDL tools during compilation, which will be overlooked otherwise.

3.4 Interface Model Extractor

To effectively generate test cases, FANS will extract interface models of target services. Here, we briefly introduce the design principles and design choices of interface model extraction, then detail how to extract the interface model, including transaction code, input and output variables, as well as type definitions.

3.4.1 Principles of Extraction

Three principles are recommended when designing the interface model extractor:

Complete: As we want to fuzz Android native system services systematically, we need to obtain a complete set of interfaces, together with all transactions of them. All of the interfaces have been collected by interface collector.

Precise: Since the target interfaces will fall back on exception handling when invalid random inputs are given in the transaction request, we need a precise interface model to generate valid inputs that pass sanity checks. We handle the precision of the model from the following aspects: *variable patterns*, *variable names* and *variable types*. The variable pattern implies input formats, as will be discussed later. The other two aspects help generate semantically correct inputs.

Convenient: Ideally, a convenient method should be adopted for interface model extraction. Besides, we had better find a unified approach to handle both the interfaces defined in C++ and those defined in AIDL.

3.4.2 Design Choices of Extractor

With the above principles in mind, we have made the following design choices for the extractor:

Extract from Server Side Code: In Android, client apps call target transactions with the RPC interface `transact`. The service, i.e., the server side, handles the RPC with the `onTransact` method. This correlation means that we can extract all possible transactions on either side. We prefer to analyze the server side for the following two reasons: (1) It is service that we are to fuzz, and directly dealing with the server side will give us a more accurate view of what inputs the server-side code expect, as well as how services use inputs deserialized from `data` and outputs serialized into `reply`. (2) An interface has multiple transactions, whose definitions and implementations are in general closely distributed in the server-side code. On the other hand, client-side code may invoke them in a scattered way, causing trouble for interface model extraction.

Extract from the AST Representation: There are many representations of the code. We have to choose a proper one to base the analysis on. First, since some interfaces are defined in AIDL, a candidate solution is to extract the interface model from AIDL files. However, this method will miss a wide range interfaces directly implemented in C++ in the Android source code. We can convert files of one format to another format to address this issue. Here we choose to convert AIDL files to C++ files because: (1) Existing AIDL tools can generate C++ implementations of interfaces defined in AIDL files without losing information. (2) Converting C++ implementations to AIDL files is not trivial and might decrease the precision of the interface model. It may lose some important information, when, for example, a variable is available under a specific path condition.

After converting AIDL files to C++ files, another choice is to extract the interface model from an intermediate representation (IR), e.g., the LLVM IR provided by the Clang compiler. But IRs usually optimize out some information, e.g., type aliases, making it harder to extract precise interface information.

On the other hand, the AST is a good representation for interface model extraction. In the AST, variable names and variable types are kept intact. Also, every type cast expression is recorded in AST. In addition, the compiler resolves all header file dependencies and provide types in correct order in the AST. Thus, we can process the AST sequentially to resolve the original type of a `typedef` type. Besides, the AST provides a clear view of all transaction codes of each interface in the `onTransact` dispatcher, as shown in Figure 2. Lastly, each statement (e.g., sequential statement and conditional statement) is separated in the AST. These characteristics make it convenient to extract the interface model from the AST representation.

3.4.3 Transaction Code Identification

As described in Section 2.1, the `onTransact` function in a target interface dispatches the control flow to target transactions according to the transaction `code`. This dispatch process is usually implemented as a `switch` statement in the C++ source, and converted to multiple `case` nodes in the AST, where each `case` represents a transaction to invoke. Therefore, we can readily identify all transactions of a target interface by analyzing *case* nodes in the AST and recognize the associated constant transaction `code`.

3.4.4 Input and Output Variable Extraction

After identifying transaction codes, we need to extract inputs deserialized from the `data` parcel in each transaction. Besides, as we would like to infer inter-transaction dependencies, we also need to extract transactions' outputs which are serialized into the `reply` parcel.

Specifically, there are three possible classes of variables used in a transaction:

- **Sequential Variables.** This type of variables exists without any preconditions.

```
1  // checkInterface
2  CHECK_INTERFACE(IMediaExtractorService, data,
       reply);
3  // readXXX
4  String16 opPackageName=data.readString16();
5  pid_t pid = data.readInt32();
6  // read(a,sizeof(a)*num)
7  effect_descriptor_t desc = {};
8  data.read(&desc, sizeof(desc));
9  // read(a)
10 Rect sourceCrop(Rect::EMPTY_RECT);
11 data.read(sourceCrop);
12 // readFromParcel
13 aaudio::AAudioStreamRequest request;
14 request.readFromParcel(&data);
15 // callLocal
16 callLocal(data, reply, &ISurfaceComposerClient::
       createSurface);
17 // function call
18 setSchedPolicy(data);
```

Listing 1: Sequential Statement Example

- **Conditional Variables.** This type of variables depends on some conditions. If these conditions are not satisfied, the variables could be `NULL` or do not appear in the data, or even have a different type than when the conditions are satisfied.
- **Loop Variables.** This type of variables are deserialized in loops, and even nested loops.

These three types of variables correspond to three types of statements in the program exactly, i.e., sequential statement, conditional statement and loop statement. As a result, we will mainly process these kinds of statements in the AST. Besides, we will also consider the return statement. The reason will be detailed in the corresponding part. Moreover, as `onTransact` function processes inputs and outputs similarly, we only demonstrate the details with input variables.

A. Sequential Statement: As shown in Listing 1, there are mainly seven kinds of sequential statements:

(1) **checkInterface.** The server will check the interface token (unique for every interface) given by the client at the beginning of each transaction. If the interface token does not match, it will just return, which suggests that we cannot fill random bytes into `data` parcel.

(2) **readXXX.** In Line 4, `readString16` deserializes a common type, i.e., `String16`, from the `data` parcel. The variable name also holds some semantics. In this case, the `opPackageName` should be a package name. Besides, in Line 5, `readInt32` reads a `int32_t` variable, while the left-hand-side variable type is `pid_t`. In such a case, we will always choose the type with richer semantics as the variable type, i.e., `pid_t`. We will also apply this strategy to type cast expressions.

(3) **read(a, sizeof(a) * num).** In this circumstance, the server will directly copy a raw structure or an array from the data parcel. In Line 8, the server reads a structure whose type is `effect_descriptor_t`.

(4) **read(a).** Here, the server will read a Flattenable or Light-Flattenable structure. In Line 11, the server reads a Light-Flattenable structure `Rect`.

(5) **readFromParcel.** This kind of sequential statement is special in that the deserializtion process happens in another class or structure which implements the `Parcelable` interface. In Line 14, the server reads a class whose type is `aaudio::AAudioStreamRequest`.

(6) **callLocal.** Taking Line 16 as an example, `callLocal` method will process the arguments of `createSurface` one by one. If the variable type is not a pointer, it is considered as an input variable. Otherwise, it is considered as an output variable.

(7) **Misc Function.** For those special input formats, the `data` parcel will be passed into a function. In Line 18, the `data` parcel is passed into the function `setShedPolicy`. For such a case, we will mark this input as a function and recursively handle the data. Moreover, this indicates we should also collect the file which includes the corresponding function, e.g., `setSchedPolicy` in this case.

```
1 int32_t isFdValid = data.readInt32();
2 int fd = -1;
3 if (isFdValid) {
4     fd = data.readFileDescriptor();
5 }
```

Listing 2: Conditional Statement Example

B. Conditional Statement: There are several kinds of conditional statements, e.g., `if` statement and `switch` statement. Here we demonstrate our approach to the `if` statement. As shown in Listing 2, whether Line 4 will be executed or not is decided by the `isFDValid` variable. In such a case, we consider `fd` as a *conditional input*. Besides, we record the condition for `fd` to get a more precise interface model.

C. Loop Statement: There are several forms of *loop* statements, e.g., `for` statement and `while` statement. Here we demonstrate our approach to the `for` statement. As shown in Listing 3, we record the number of times `key` is read, i.e., `size`. We consider `key`, `fd` and `value` as *loop variables*. Moreover, there might be a kind of `for` statement,

```
1  const int size = data.readInt32();
2  for (int index = 0;index<size; ++index){
3      ...
4      const String8 key(data.readString8());
5      if(key == String8("FileDescriptorKey")){
6          ...
7          int fd = data.readFileDescriptor();
8          ...
9      } else {
10         const String8 value(data.readString8());
11         ...
12     }
13 }
```

Listing 3: Loop Statement Example

```
1 const uint32_t numBytes=data.readInt32();
2 if(numBytes>MAX_BINDER_TRANSACTION_SIZE){
3   reply->writeInt32(BAD_VALUE);
4   return DRM_NO_ERROR;
5 }
```

Listing 4: Return Statement Example

`for(auto i: vector)`, which does not explicitly declare the cycle count. We heuristically guess that the cycle count is the previous value read from the parcel before the `for` statement, e.g., `size` in Line 1. Furthermore, we can observe that there is also a conditional statement, which implies that these types of statements can be nested together.

D. Return Statement: *Return* statement is special among these statements. During a transaction, several *return* statements might appear, which lead to different execution paths. If a path returns an error code, it implies that this path is less likely to have vulnerabilities. Thus, we will assign this path a low probability, which means that fewer test cases taking this path will be generated. As Listing 4 shows, if `numBytes` is larger than `MAX_BINDER_TRANSACTION_SIZE`, the function will simply return an error code `DRM_NO_ERROR`. In such a case, we should try not to generate a value larger than `MAX_BINDER_TRANSACTION_SIZE` when generating `numBytes`. Besides, it will also help us generate explicit inter-transaction dependency, as inputs that do not satisfy the dependency usually fall back to error handling paths.

3.4.5 Type Definition Extraction

Apart from extracting input and output variables in transactions, we also extract type definitions. It helps enrich the variable semantics so as to generate better inputs. There are three kinds of types to analyze:

- **Structure-like Definition.** This kind of types includes union and structure. We could easily extract the member of these kinds of objects from the AST.
- **Enumeration Definition:** As for enumeration type, we should extract all given (constant) enumeration values.

```
1  typedef int __kernel_pid_t;
2  typedef __kernel_pid_t __pid_t;
3  typedef __pid_t pid_t;
4  typedef struct effect_descriptor_s {
5      effect_uuid_t type;
6      effect_uuid_t uuid;
7      uint32_t apiVersion;
8      uint32_t flags;
9      uint16_t cpuLoad;
10     uint16_t memoryUsage;
11     char     name[EFFECT_STRING_LEN_MAX];
12     char     implementor[EFFECT_STRING_LEN_MAX];
13 } effect_descriptor_t;
```

Listing 5: Typedef Statement Example

- **Type Alias:** There are many `typedef` statements in AOSP. As shown in Listing 5, `pid_t` is actually an `int` type. As a result, we could generate variables of type `pid_t` with random integers. Besides, `effect_descriptor_t` in Listing 1 is actually `struct effect_descriptor_s`. Without such typedef knowledge, we could not generate semantics-rich inputs.

Also, as AOSP is a monolithic project, we need to add the namespace to variable types so as to avoid conflicts when extracting these kinds of type knowledge. Besides, guaranteed by the compiler, all headers used by the C/C++ files will be included in AST in order. As a result, we can collect definitions of all related types.

3.5 Dependency Inferer

After extracting interface models, we infer two kinds of dependencies: (1) *interface dependency*. That is, how a multi-level interface is recognized and generated. It also implies how an interface is used by other interfaces. (2) *variable dependency.* There are dependencies between variables in transactions. Previous researches rarely consider these dependencies.

3.5.1 Interface Dependency

In general, there are two types of dependencies between interfaces, corresponding to the generation and use of interfaces.

Generation Dependency If an interface can be retrieved via another interface, we say that there is a generation dependency between these two interfaces. As introduced in Section 2.1, we can get Android native system service interfaces, i.e., top-level interfaces, directly from the service manager. As regards multi-level interfaces, we find that upper-level interface will call `writeStrongBinder` to serialize a deep interface into `reply`. In this way, we can easily collect all generation dependencies of interfaces.

Use Dependency If an interface is used by another interface, we say that there is a use dependency between these two interfaces. We find that when an interface A is used by another interface B, B will call `readStrongBinder` to deserialize A from `data` parcel. Hence, we can utilize this pattern to infer the use dependency.

3.5.2 Variable Dependency

There are two types of variable dependencies, i.e., intra-transaction and inter-transaction dependency, based on whether the variable pair is in a same transaction.

Intra-Transaction Dependency One variable sometimes depends on another in the same transaction. As demonstrated in Section 3.4.4, there could be conditional dependency, loop dependency, and array size dependency between variables in a transaction. Conditional dependency refers to the case where the value of one variable decides whether another exists or

Algorithm 1 Inference of Inter-Transaction Dependency

```
Input: Interface Model (M)
Output: Inter-Transaction Dependency Graph (G)
G = {}
I = [ ] // input variables
O = [ ] // output variables
for variable in M do
    if variable is input then
        add variable into I
    end if
    if variable is output then
        add variable into O
    end if
end for
for iVar in I do
    for oVar in O do
        if iVar.txID != oVar.txID then
            if iVar.type==oVar.type then
                if iVar.type is complex then
                    add edge (iVar, oVar) into G
                else if iVar.name and oVar.name are similar then
                    add edge (iVar, oVar) into G
                end if
            end if
        end if
    end for
end for
```

not. For example, `fd` in Listing 2 conditionally depends on `isFdValid`. Loop dependency refers to the case where one variable decides the number of times another is read or written, as the variables `size` and `key` in Listing 3. For the last one, the size of an array variable is specified by another variable. When generating this array variable, we should generate the specified number of items.

Inter-Transaction Dependency A variable sometimes depends on another variable in a different transaction. In other words, input in one transaction can be obtained through output in another transaction. We propose Algorithm 1 to deal with this kind of dependency. Specifically, we extract the inter-transaction dependencies following the principles below: ① one variable is input, and the other is output; ② these two variables are located in different transactions; ③ input variable's type is equal to the output variable's type; ④ either the input variable type is complex (not primitive type), or the input variable name and the output variable name are similar. The similarity measurement algorithm can be customized.

3.6 Fuzzer Engine

After inferring the dependencies, we can start fuzzing Android native system services. Firstly, the fuzzer manager will sync the fuzzer binary, interface model, and dependencies to mobile phone and start the fuzzer on the smartphone. Then the fuzzer will generate a test case, i.e., a transaction and its corresponding interface to fuzz the remote code. Besides, the *fuzzer manager* will sync the crash logs from smartphones regularly. Here we mainly demonstrate the test case generator, as other parts are straightforward in FANS. Interested readers could refer to the source code we open source for details.

When fuzzing Android native system services, we are fuzzing the transaction specified by the transaction code. Therefore we can randomly generate a transaction at first and then invoke its corresponding interface.

Transaction Generator We can generate input variables of a transaction one by one based on the interface model. During the generation, we follow the principles in order as below.

- **Constraint First.** If a variable is constrained by another variable, we should check the constraints before generating the variable. For instance, as shown in Listing 2, `isFDValid` should be checked before generating `fd`.
- **Dependency Second.** If a variable can be generated by other transactions, we should use them to generate it with a high probability. In such circumstances, we should generate the dependent transaction first, and then get the output from the corresponding `reply` parcel. Also, we do not follow this principle for a low probability.
- **Type and Name Third.** We may generate a variable according to its type and name, no matter the aforementioned dependencies exist or not. For example, in Listing 1, we will generate a valid package name (`String16`) for `opPackageName`. Besides, we will generate a valid process ID (`int`) for `pid`. For a complex type, we will generate its members recursively according to this rule.

Interface Acquisition As for top-level interfaces, we can get them through the service manager. Multi-level interfaces can then be recursively obtained via the recognized interface dependency.

4 Implementation

We implemented a prototype of FANS from scratch, rather than developing one based on an existing fuzzer, e.g., AFL [25], for the following reasons. First, it takes huge engineering work to port AFL to Android. Second, AFL-based fuzzers are effective at testing one standalone program or service, thus we have to compile and test each target service one by one, which is non-scalable. Third, AFL is not effective at testing service-based applications, including the binder IPC based services. Table 1 shows the statistics of this implementation.

Interface Collector To be able to collect interfaces efficiently, we first compile the AOSP codebase, recording the compilation commands in the meantime. Then we walk these commands while scanning for the characteristics pointed out

Table 1: Implementation Details of FANS

Component	Language	LoC
Interface Collector	Python	145
Interface Model Collector	C++, Python	5238
Dependency Inferer	Python	291
Fuzzer Engine	C++, Python	5070
Total	C++, Python	10744

in Section 3.3 and Section 3.4.4. This step can be easily implemented with Python.

Interface Model Extractor As we are extracting interface models from AST, we first convert the compilation commands to *cc1* commands while linking with the Clang plugin which is used to walk the AST and extract a rough interface model. We do an approximate slice on the AST and only preserve statements relevant to input and output variables, omitting others. Finally, we do a post-process on the rough model so that *fuzzer engine* can easily use it. The interface model is stored in JSON format.

Dependency Inferer Given the interface model described with JSON, dependency inferer traverses the model and makes interface dependency inference as explained in Section 3.5.1. Besides, dependency inferer will also get the inter-transaction dependency according to Algorithm 1.

Fuzzer Engine We implement a simple fuzzer manager so as to run fuzzer on multiple phones together with syncing data between host and smartphones. We build the entire AOSP with ASan enabled. The fuzzer is implemented in C++ as a native executable. As some Android native system services check the caller's permission when receiving RPC requests, the fuzzer is executed under root privilege. To accelerate the execution, we always make asynchronous RPCs through marking the `flag` argument of `transact` as 1 when the outputs in `reply` are not needed. When we do need the outputs in `reply`, e.g., dependency inference, we make synchronous calls. Finally, in order to analyze triggered crashes, we use the builtin logcat tool of Android for logging. Besides, we will also record native crash logs located in `/data/tombstones/`.

5 Evaluation

In this section, we evaluate FANS to answer the following questions:

(1) How many interfaces have been found? What is the relationship between them? (Section 5.1)

(2) What does the extracted interface model look like? Is the model complete and precise? (Section 5.2)

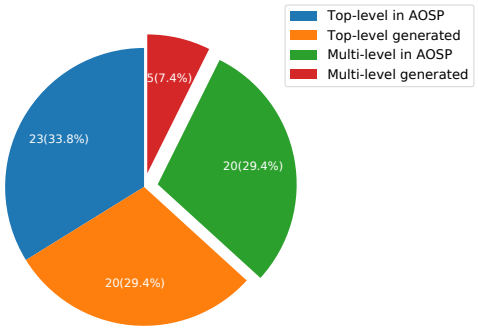

Figure 3: Interface Statistics: 43 top-level Android native system interfaces are discovered, of which 23 are from AOSP and 20 are generated from AIDL files. 25 multi-level Android native system interfaces are discovered, of which 20 are from AOSP and 5 are generated from AIDL files.

(3) How effective is FANS in discovering vulnerabilities of Android native system services? (Section 5.3)

Experimental Setup As shown in Figure 2, we implement the first three components on Ubuntu 18.04 with i9-9900K CPU, 32 GB memory, 2.5 T SSD. As for test devices, we use the following Google's Pixel series products: Pixel * 1, Pixel 2XL * 4, and Pixel 3XL * 1. We flash systems of these smartphones with AOSP build number `PQ3A.190801.002`, i.e., `android-9.0.0_r46`, which is a recent version supporting these devices when writing this paper. Although the Android release versions are the same, the source code can be slightly different for different Pixel models. For the following two sections (Section 5.1, Section 5.2), we report the experiment results carried out on Pixel 2XL.

5.1 Interface Statistics and Dependency

In this section, we systematically analyze the interfaces collected by the interface collector and introduce the dependencies among these interfaces.

5.1.1 Interface Statistics

It takes about an hour to compile AOSP. However, it only takes a few seconds to find the interfaces in the source code. As shown in Figure 3, multi-level interfaces account for as many as 37% of all native service interfaces, which highlights the necessity to examine more interfaces than registered at the service manager. Besides, interfaces generated by AIDL tools also take a large part, so we should extract interfaces directly inside AOSP and interfaces generate from AIDL files. We are not able to compare the number of interfaces discovered by FANS with any other existing research, as none ever focused on Android native system services.

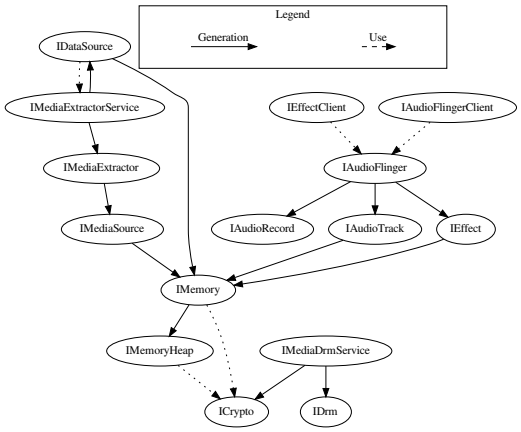

Figure 4: Part of the Interface Dependency Graph

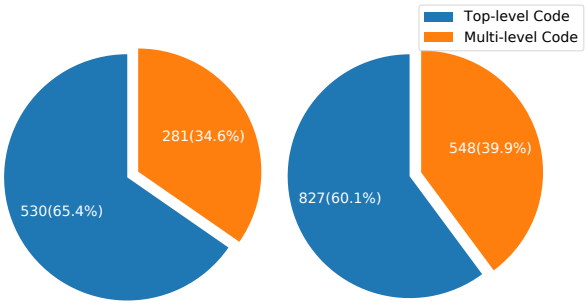

Figure 5: Transaction Details in Interface: 530 top-level transactions and 281 multi-level transactions are found. 827 top-level transaction paths and 548 multi-level transaction paths are found.

5.1.2 Interface Dependency

It just takes seconds to infer the interface dependency relationship. As the full interface dependency graph is too large (see Figure 8 in Appendix), we demonstrate the complexity of interface dependency with one of its representative parts, as shown in Figure 4. The deepest interface is `IMemoryHeap`, whose ancestor is `IMediaExtractorService`. It requires five steps to get the `IMemoryHeap` interface. Without dependency relationships, we could not obtain such a deep interface easily and automatically. It also comes to our notice that a multi-level interface can be obtained from several upper interfaces. For example, `IMemory` can be obtained from the `IMediaSource`, `IEfect`, and `IAudioTrack` interfaces. Therefore, we can explore different paths to fuzz a same interface. Besides, there are some other interfaces which are neither top-level interfaces nor multi-level interfaces, but the architecture remains the same. We call such interfaces customized interfaces. Customized interfaces are designed to customize system functionality as needed and can be manually instantiated by developers and passed to top-level or multi-level interfaces. For example, `IEffectClient` interface is transferred to some transaction A of `IAudioFlinger`. Transaction A will call the method provided by the `IEffectClient` interface later. To the best of our knowledge, we are the first to systematically investigate the dependencies between the interfaces in Android native system services.

5.2 Extracted Interface Model

The process of extracting a rough interface model takes about an hour. The post-process of the interface model extractor only takes seconds. We also give the time for inferring the variable dependency as follows. The time used to infer intra-transaction dependencies has already been counted into that of extracting the interface model. As to the time for inter-transaction dependency inference, it is also a matter of seconds.

We start this section by discussing the extracted interface model statistics, and then talk about the completeness and precision of the interface model.

5.2.1 Extracted Interface Model Statistics

We discuss the extracted interface model from two aspects: transaction and variable.

Transaction As shown in Figure 5, there are 811 transactions inside the Android native system services, in which multi-level transactions account for 281, a proportion of about 35%. Besides, in either top-level interfaces or multi-level interfaces, the transaction path quantity is over 1.5 times that of the transaction, which means many transactions hold more than one return statement in the sliced AST. In other words, if we do not distinguish between different transaction paths, we cannot obtain an explicit dependency since some inter-transaction dependencies only exist on a particular path.

Variable We only count in variables that are directly inside `onTransact`. That is, we do not count variables recursively. For instance, `onTransact` uses `readFromParcel` to read a structure. It is only in `readFromParcel` that the structure's members are dealt with, so we exclude them from the statistics. Otherwise, the statistics would be imprecise. As shown in Figure 6, there are various types as described in Section 3.4.4, e.g., structure and file descriptor. We explain the figure from three aspects: variable patterns, type aliases, and inter-transaction variable dependencies.

- **Variable Pattern.** According to variable patterns, we divide variables into three kinds as demonstrated in Section 3.4.4: sequential variable, conditional variable and loop variable. We notice that few variables are in simple sequential statements, and most variables processed in sequential statements have `String` type. The reason behind this is that nearly all interfaces check

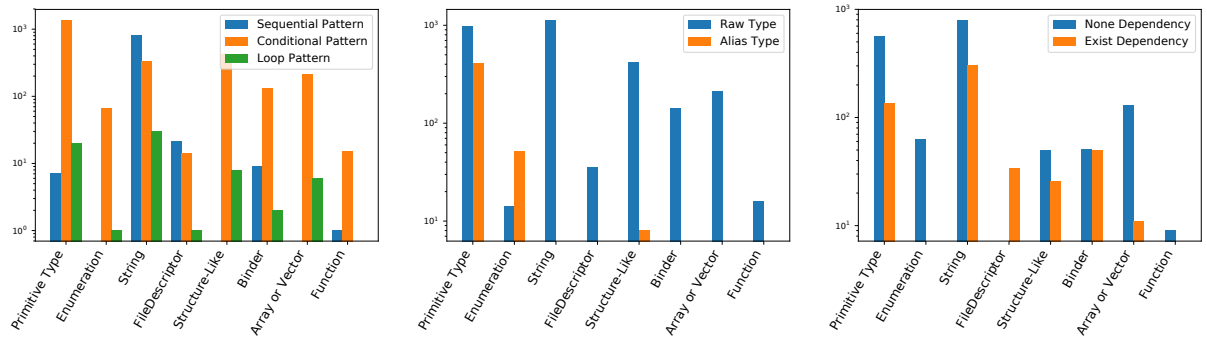

Figure 6: Classification Result of Variables by Variable Pattern, Type Alias, and Dependency

the interface token at the beginning of each transaction except several `SHELL_COMMAND_TRANSACTION` transactions and the only one `GET_METRICS` transaction in the `IMediaRecorder` interface. In other words, almost all variables are conditional variables. Therefore, we have to extract the constraints imposed on variables to generate valid inputs. Constraint extraction is also necessary for solving intra-transaction dependencies. Additionally, it is possible for almost all variable types to occur in a loop.

- **Type Alias.** As for type alias, i.e., type defined in `typedef` statement, we notice that all aliases are for three types: primitive types, enumeration types, and structure-like types. This makes sense as we usually use `typedef` statements for more semantic types, which can be seen from List 5. By all means, we would lose semantic knowledge of variables without these typedef statements.
- **Variable Dependency.** Here we consider inter-transaction dependencies. Since there is no dependency on output variables, we focus on input variables. Moreover, we generate array dependency according to the array item type. As shown in Figure 6, there are dependencies among almost all variable types, in particular primitive types and the string type. Besides, structure-like and binder-type variables can also be generated based on dependency, which helps generate more semantic and well-structured inputs, resulting in deep fuzzing into Android native system services.

5.2.2 Completeness and Precision of Extracted Interface Model

As there is no ground truth about the interface model, we randomly select ten interfaces and manually check whether the extracted model is complete and precise according to the principle mentioned in Section 3.4.1. We find that we successfully recover all the transaction codes, fulfilling completeness. Almost all variable patterns, variable names and variable types are recovered as well. In conclusion, the model is not entirely precise but good enough. What's more, inter-transaction variable dependencies are calculated with Algorithm 1 in Section 3.5.2.

As far as we know, no previous work focuses on Android native system services, precluding any comparison. However, we argue that most existing researches cannot handle Android native system services effectively. Chizpurfle [10] focuses on vendor-implemented Java services and cannot deal with Android native system services. BinderCracker [6] tests all services in Android but is unable to infer a more complete and precise model than FANS when applied to Android native system services. This is due to the fact that BinderCracker is based on app traffic, which might miss rarely used RPCs and lose various variable semantics like variable names and types.

5.3 Vulnerability Discovery

To evaluate how effective FANS is, we intermittently ran FANS on our six smartphones for around 30 days. However, we were not able to get the precise run-time of FANS during the 30 days' experiment due to the following reasons: (1) The fuzzer might crash every several minutes. (2) As we ran the experiment on real machines, once the Android system crashed, we had no choice but to re-flash them manually. Moreover, the device could enter recovery mode even when the fuzzer had started less than ten minutes ago. These situations decreased the fuzzing efficiency and also prevented collecting statistics about run-time. Despite this, we have discovered 30 unique bugs from thousands of crashes reported by FANS.

All of the 30 vulnerabilities are listed in Table 2. Apart from the 22 vulnerabilities found in Android native system services, there are five vulnerabilities in the libraries `libcutils.so`, `libutils.so` and `libgui.so`, which are used as public libraries in Android native system services. Furthermore, we found three vulnerabilities in Linux system components. For instance, we discovered a stack overflow in `iptables-restore`. This program is a user-space program for firewall configuration provided by Linux kernel. These vulnerabilities prove that inputs generated by FANS can drive the control flow into deep paths under complicated constraints.

Table 2: Vulnerabilities found by FANS

	Component	Vulnerability File (binary or so)	AndroidID	Vulnerability Type	Status
1		libsensor.so	-	Heap user after free	Reported
2		libsensor.so	128919198	Out of Memory	Confirmed
3		libsensor.so	128919198	Out of Memory	Confirmed
4		libsensor.so	-	Assertion failure	Reported
5		libsensorservice.so	143896234	Illegal fd	Confirmed
6		libmediadrm.so	143897317	new_capacity overflow	Confirmed
7		libmediadrm.so	143895981	new_capacity overflow	Confirmed
8		libmediadrm.so	143896237	Null pointer dereference	Confirmed
9		libmediametrics.so	143896917	Null pointer dereference	Confirmed
10		libsurfaceflinger.so	143899028	invalid memory access	Confirmed
11	Android Native	libsurfaceflinger.so	143897162	invalid memory access	Confirmed
12	System Service	libaaudioservice.so	143895840	Null pointer dereference	Confirmed
13		libaudiopolicymanagerdefault.so	-	key not found	Reported
14		libmediaplayerservice.so	-	CHECK failure	Reported
15		installd	143899228	Stack buffer overflow	Confirmed
16		installd	143898908	incomplete check	Confirmed
17		installd	-	CHECK failure	Reported
18		installd	-	CHECK failure	Reported
19		statsd	143897309	Null pointer dereference	Confirmed
20		statsd	143895055	Out-of-bound access	Confirmed
21		incidentd	143897849	Null pointer dereference	Confirmed
22		gatekeeperd	143894186	Null pointer dereference	Duplicated
23		libcutils.so	143898908	integer overflow	Confirmed
24		libcutils.so	143898343	Null pointer dereference	Confirmed
25	Basic Library	libutils.so	-	integer overflow	Reported
26		libgui.so	-	mul-overflow	Reported
27		libgui.so	-	Null pointer dereference	Reported
28		iptables-restore	143894992	Stack buffer overflow	Duplicated
29	Linux Component	ip6tables-restore	143895407	Stack buffer overflow	Duplicated
30		fsck.f2fs	-	heap-buffer-overflow	Reported

Moreover, although we aim to discover vulnerabilities in Android native system services implemented in C++, we triggered 138 Java exceptions, such as `FileNotFoundException`, `DateTimeException`, `NoSuchElementException`, and `NullPointerException`. This can be attributed to the fact that Java applications sometimes depend on Android native system services. Some native services also invoke Java methods. Since robustness and stability are important for Android native system services, these Java exceptions should not have occurred. Stricter checks should be enforced to solve this problem.

We have reported all native vulnerabilities to Google. 20 of them were confirmed and 18 Android IDs were given, three of which are duplicate with undisclosed vulnerability report. Up to now, Google has assigned moderate severity to Android ID 143895055 and 143899228. Google has also assigned CVE-2019-2088 to Android ID 143895055 and will put us in their acknowledgment page in the future. Submission of Java exceptions is in progress.

Comparison with Existing Research It is not trivial work to compare our solution with related work. To the best of our knowledge, BinderCracker [6] is the most relevant one. BinderCracker works on Android system services before Android 6.0, including Java system services and native system services. However, Android began to support clang only after Android 7.0. As we utilize an LLVM plugin to extract the interface model, it is not easy to port our approach to lower Android versions. Besides, BinderCracker is closed-source, so we cannot test it on modern Android, e.g., android-9.0.0_r46. Moreover, BinderCracker did not show detailed vulnerability types. We are thus forced to a simple comparison of the number of vulnerabilities discovered by the two tools. BinderCracker found 89 vulnerabilities on Android 5.1 and Android 6.0, both native vulnerabilities and java exceptions included. Although we only focus on Android native system services, we found 30 native vulnerabilities and 138 Java exceptions, way more than 89. We believe this comparison is convincing that FANS is superior over BinderCracker as Android security has been improving over the years.

5.4 Case Studies

We present three vulnerabilities discovered by FANS. Firstly, we look into the root causes of these vulnerabilities and demonstrate how to trigger vulnerabilities. Also, we explain how design choices (e.g., categorizing variables as sequential, conditional, and loop ones) help generate inputs that trigger vulnerabilities. Secondly, we show our insights into these vulnerabilities and devise mitigation for them.

5.4.1 Case Study I: new_capacity overflow Inside readVector of IDrm

Attack There are multiple new_capacity overflow vulnerabilities in `IDrm`, a second-level interface obtained via `IMediaDrmService`. The bugs are all triggered by the same function, `BnDrm::readVector`. The function invokes `insertAt` to allocate a buffer whose size is decided by the variable `size` in `data`. Inside `insertAt`, there is a sanity check on the insertion index, which will return `BAD_INDEX` in case of a lousy index. However, no check is made on the `size` argument. According to the interface dependency graph, FANS could generate `IDrm` interface automatically. When it comes to the variable name `size`, FANS generates some dangerous values, e.g., -1, which can easily trigger the vulnerability. We could further achieve DoS attack through this kind of vulnerability, preventing other apps from using necessary services.

Insight Buffer allocation is a core step in IPC, and also a very vulnerable one. Vulnerabilities can easily occur during this process if the server puts any trust in the client and skips necessary sanity checks. Unfortunately, this problem is prevalent among Android native system services and is persistent. In `BnDrm` alone, the problematic `readVector` is called for more than 30 times, making an easy target for attackers. Performing proper sanity checks would effectively mitigate this problem. Nevertheless, it is not an easy task considering the mass body of Android source codes. Fortunately, there are safely implemented deserialization functions provided by `Parcel`, which perform input validations. These standard functions are preferable to the error-prone customized functions. In this case, replacing `readVector` with `Parcel::readByteVector` would fix the vulnerability neatly.

5.4.2 Case Study II: Out-of-bound Access Inside informAllUidData of statsd

Attack The native system service `statsd` is a daemon in Android 9. In the transaction `Call::INFORMALLUIDDATA`, `statsd` deserializes three vectors from `data` parcel containing items of `int32_t`, `int64_t` and `::android::String16` respectively. These vectors are passed into `informAllUidData` and then forwarded to the `updateMap` method of `UidMap`. Function `updateMap` iterates on the three vectors in a loop. The size of vector `uid` out of the three is used as the loop count. Since items in any one vector are supposed to have a one-to-one correspondence to those in the other two vectors, the three vectors are expected to have the same length, so that the iteration can work normally. Nevertheless, this requirement is left unchecked. Out-of-bound access can then be achieved by passing in a longer vector of `uid` than the rest two. In order to generate the malformed transaction, FANS first identifies the variable types of these three inputs from AST. Then it generates these vectors one by one through (1) generating the vector size; (2) generating the corresponding number and type of elements. However, existing work like BinderCraker [6] might not be able to generate such effective inputs as it ignores the semantics of these variables.

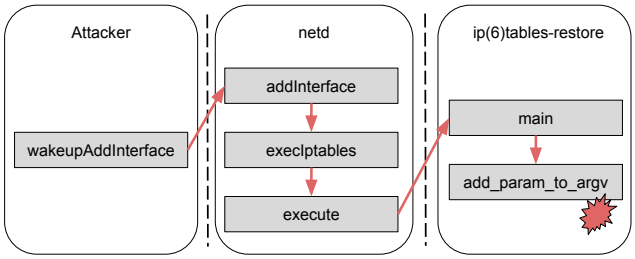

Figure 7: Call Trace of ip(6)tables-restore stack overflow

Insight In this case, the same index is used for different vectors, resulting in an OOB vulnerability. This bug, just as the last case, arises from failure in input validation. Nevertheless, it is a more interesting bug. Hopefully it can yield an exploitation other than DoS if appropriately used. This case demonstrates FANS's ability to discover meaningful bugs. Google has already fixed this vulnerability and assigned CVE-2019-2088 to us. So we do not give the mitigation here.

5.4.3 Case Study III: Stack Overflow Inside ip(6)tables-restore

Attack An unexpected stack overflow bug is found to reside in the ip(6)tables-restore binary. As we focus on Android native system services, we do not find the vulnerability directly. It is found when we fuzz the `netd` daemon, whose interface file is generated automatically. We craft `in_ifName` with a sufficiently long string quoted in the transaction `Call::WAKEUPADDINTERFACE`, then it calls `wakeupAddInterface`. Finally it triggers the stack overflow vulnerabilities in the `add_param_to_argv` function.

Figure 7 gives the detailed execution path. However, we still need to craft `in_ifName` carefully as the string is deserialized from `data` parcel through `readUtf8FromUtf16` which executes many checks. To take the last step towards successful attack, FANS tags `in_ifName` with `utf8=true` when extracting the interface model. Later FANS uses the corresponding serialization method `writeUtf8AsUtf16` to serialize `in_ifName` into `data`, which can pass the sanity checks. In contrast, BinderCracker [6] may well miss such transactions because popular apps rarely use them. Even if it could get such a transaction input format, it would randomly mutate the traffic which is likely to fail quickly in the checks mentioned above.

Insight This vulnerability crosses three processes: attacker process, `netd` and `ip(6)tables-restore`. In other words, this bug is buried deep. Furthermore, although we mainly focus on fuzzing Android native system services, we find a vulnerability in a Linux component. It suggests that there is a close relationship between Android system services and basic Linux components. In the light of this, we can assert that there is another way to fuzz Linux components. Besides, these two bugs are also present in iptable package and can be found on a regular Linux distribution. They have been fixed by the netfilter team in April 2019 and assigned CVE-2019-11360. So here we do not give the mitigation. However, at the time of writing this paper, they have not been fixed in Android.

6 Discussion

We have demonstrated FANS's effectiveness in excavating vulnerabilities in the Android native system service. Now we discuss its limitations and what we will do in the future.

Interface Model Accuracy Although we have tried our best to extract the interface models, the interface model is not perfect. For example, we assume that the loop size is the previous variable before the loop when we can not get the loop size directly. However, for loop statements that traverse a linked list, the loop size is undetermined, not as we guess. In such circumstances, we believe that it is not easy to improve it. Besides, even if a developer defines a semantic type somewhere, he might accidentally use the original type instead of the type alias. Thus we can not get a more semantic variable type, which would also affect the variable dependency generation. Other than those mentioned above, the dependency we got might be incomplete because there might exist specific order between the transaction calls as service can be seen as a state machine. However, as we are fuzzing, if we always follow the specified order, we may miss some vulnerabilities. Meanwhile, we have already found some vulnerabilities caused by incomplete state machine processing in service.

Coverage Guided Fuzzing Nowadays, coverage guided fuzzing is popular. For FANS, even though we do not use coverage knowledge of Android native system services, we find many vulnerabilities in system services audited by many experts. However, to our belief, guided with coverage, FANS can find more vulnerabilities. Moreover, as system service is state-sensitive, its coverage might be affected by inputs generated previously or by other applications' calls. This could be a challenge when integrating coverage to FANS.

Fuzzing Efficiency As some Android system services run as a daemon or might check the caller's permission, for convenience, we run fuzzer as root. Nevertheless, the root privilege is very high, which can change lots of things. During the experiment, we found that a smartphone can enter into recovery mode even just after starting the fuzzer ten minutes. As a result, we needed to flash the phone manually, which significantly affects the efficiency of FANS. We think this can be solved, either limiting the privilege of fuzzer or finding a way to flash the device automatically.

Interface-based Fuzzing in Android In Android 9, there mainly exists three kinds of services located in different domains: normal domain, vendor domain and hardware domain. In Pixel series products, applications can access only normal domain services registered in the service manager. In this paper, we mainly pay attention to the native system services in normal domain. However, these three kinds of services share the same architecture in the aspect of communication and interface implementation. Consequently, we could easily transfer the method demonstrated in this paper to other domain services, even service implemented in Java language. Besides, there also exist some similar interfaces, i.e., customized interfaces, which do not belong to the parts mentioned above. These interfaces are designed to be implemented and instantiated by applications and passed to the server-side by the clients. We can also fuzz these implementations with the methods proposed in this paper. The major drawback is that we need to instantiate these interfaces manually.

7 Related Work

IPC and Service Security in Android While the security of the Android operating system has always been the focus of academic and industrial research, similar researches for IPC and system services are deficient. In early times, vulnerable Intents were widely exploited in attacking userland applications. Therefore, the main target of the previous researches [2, 11, 16] on IPC in Android was the Intent.

Gong [7] is the first one who paid attention to the Binder IPC interface. He pointed out Binder is the actual security boundary of Android system services, and proved it insecure by discovering critical vulnerabilities manually. Wang et al. [12] further proposed a solution to fuzz Java interfaces generated from AIDL files, while Chizpurfle [10] targeted vendor implemented Java services. Further, there are some researches [3, 26] that focus on input validation vulnerabilities related to Android services. Several other researches [1, 8, 17] concentrate on the inconsistency of access control in the Android framework related to Android services.

BinderCracker [6] extends the testing to native services. It monitors the IPC traffic of several popular user apps, and tries to understand the input model and transaction dependencies through the recorded traffic, then generates new test cases accordingly. However, this solution highly depends on the diversity of the recorded traffic and is not effective. First, it cannot systematically recognize all interfaces including multi-

level interfaces to test, and cannot recognize the complete dependencies between interfaces, either. Second, the interface model and the transaction dependencies inferred from the traffic are neither (1) complete, since the traffic may overlook rarely-used transactions; nor (2) precise, since the inference is made from data which has lost many information (e.g., types).

Fuzzing for Structured Input Numerous approaches have been proposed to generate structured input for fuzzing. Generally, they fall into two categories. *Generation-based fuzzers* generates test cases from templates or predefined grammar. Peach [5] is one of the most popular fuzzer based on templates. DomFuzz [15] utilized grammar to generate dom structures for the target program. These methods suffer manual participation and poor scalability. Thus more advanced researches [9, 20, 22, 23] are proposed to handle this limitation. *Mutation-based fuzzers* mutate existing test cases to generate new ones without any input grammar or input model. VUzzer [14] runs dynamic taint analysis (DTA) to capture common characteristics of valid inputs. TaintScope [21] uses DTA to identify the checksum field. T-Fuzz [13] also bypasses sanity checks and fuzzes the guarded codes directly. Some recent fuzzing tools [18, 24, 27], referred to as hybrid fuzzers, combine fuzzing with concolic execution. This may be a promising way of fuzzing programs with structured inputs.

8 Conclusion

In this work, FANS is designed to meet the challenges in fuzzing Android native system services. Experiments have validated its ability to automatically generate transactions and invoke the corresponding interface, which greatly helps to fuzz Android native system services. Our evaluation shows that FANS is also capable of inferring the complex dependencies between these interfaces. Moreover, we discover that the interface model is very complex in three aspects: variable pattern, type alias and variable dependency. We intermittently ran FANS on our six smartphones for around 30 days and reported 30 native vulnerabilities to Google, of which 20 have been confirmed. These vulnerabilities imply that without a precise interface model, we could not fuzz Android native system services deeply. Surprisingly, 138 Java exceptions were also exposed, which may deserve further study.

Acknowledgement

We would like to thank all anonymous reviewers and our shepherd, Dr. Manuel Egele, for their valuable feedback that greatly helped us improve this paper. Besides, we would like to thank Xingman Chen, Kaixiang Chen, Zheming Li for revising the draft of this paper. This work was supported in part by National Natural Science Foundation of China under Grant 61772308, 61972224, U1736209 and U1936121, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *NDSS*, 2018.

[2] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679. IEEE, 2015.

[3] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 361–370. ACM, 2015.

[4] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[5] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, 34, 2011.

[6] Huan Feng and Kang G Shin. Understanding and defending the binder attack surface in android. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 398–409. ACM, 2016.

[7] Guang Gong. Fuzzing android system services by binder call to escalate privilege. *BlackHat USA*, 2015, 2015.

[8] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android's middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 25–36, 2019.

[9] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[10] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11. IEEE, 2017.

[11] Fauzia Idrees and Muttukrishnan Rajarajan. Investigating the android intents and permissions for malware detection. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 354–358. IEEE, 2014.

[12] Wang Kai, Zhang Yuqing, Liu Qixu, and Fan Dan. A fuzzing test for dynamic vulnerability detection on android binder mechanism. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 709–710. IEEE, 2015.

[13] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[14] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[15] Jesse Ruderman. Releasing jsfunfuzz and domfuzz, 2015.

[16] Raimondas Sasnauskas and John 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)*, pages 1–5. ACM, 2014.

[17] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[18] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[19] Dmitry Vyukov. Syzkaller, 2015.

[20] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[21] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[22] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[23] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[24] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[25] Michal Zalewski. American fuzzy lop. *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[26] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating insecure input validations in android services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1178. ACM, 2018.

[27] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

A Appendix

A.1 Full Interface Dependency Graph

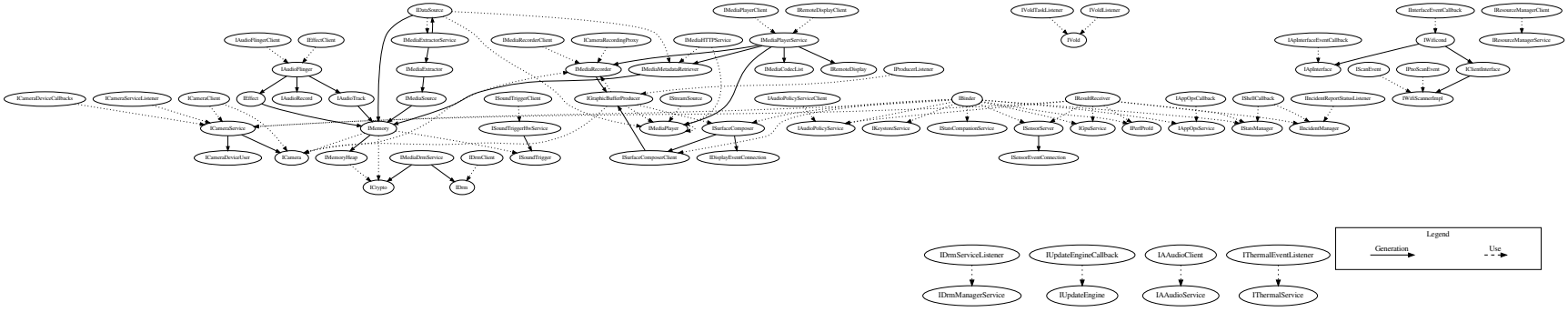

Figure 8: Full Interface Dependency Graph