Exploit the Last Straw That Breaks Android Systems

Lei Zhang[1], Keke Lian[1], Haoyu Xiao[1], Zhibo Zhang[1], Peng Liu[2], Yuan Zhang[1], Min Yang[1], Haixin Duan[3]
1: Fudan University, 2: The Pennsylvania State University, 3: Tsinghua University
1: {zxl, kklian20, hyxiao20, zbzhang15, yuanxzhang, m_yang}@fudan.edu.cn
2: pxl20@psu.edu, 3: duanhx@tsinghua.edu.cn

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833563

***Abstract*—The Android system services usually play a critical role in running multiple important tasks, and delivering seamless user experiences, e.g., conveniently storing user data. In this paper, we conduct the first systematic security study on the data storing process in Android system services, and consequently discover a novel class of design flaws (named Straw), which can lead to serious DoS (Denial-of-Service) attacks, e.g., *permanently* crashing the whole victim Android device.**

Then we propose a novel directed fuzzing based approach, called StrawFuzzer, to automatically vet all system services against the straw vulnerabilities. StrawFuzzer balances the trade-off between path exploration and vulnerability exploitation. By applying StrawFuzzer on three Android systems with the latest security updates, we identified 35 unique straw vulnerabilities affecting 474 interfaces across 77 system services and successfully generated corresponding exploits, which can be used to conduct various permanent/temporary DoS attacks. We have reported our findings with suggestions for repairing the vulnerabilities to corresponding vendors. Up to now, Google has rated our vulnerability as high severity.

I. Introduction

In Android, many system services play a critical role in running important tasks, especially storing user and system data. For example, the system service *"AccountManagerService"* can help apps save user account information, letting users conveniently avoid repeated login whenever the apps are opened.

Recently, the security of system services has attracted more and more attention. However, up to now, their data storing process is rarely understood and sanitized. In this work, we conduct the first systematic security study on it. As a result, we find that there are several different types of critical data storing operations and instructions which are frequently used, yet largely unprotected inside system services. More specifically, many critical data storing instructions are exposed to untrusted third-party apps through the services' public interfaces. An untrusted third-party app can send well-crafted messages to the target system service, access its data storing instructions, and inject trash data into the corresponding memory objects. After the cumulative efforts of the attack (e.g., repeating a number of times), all memory resources (e.g., heap) are exhausted. Finally, this attack can crash the victim system service and break down the Android device (e.g., rebooting).

Additionally, we find some of the above DoS (denial of service) attacks cause a permanent result. In some cases, the victim device cannot be recovered from rebooting once attacked. Figure 1 illustrates an example of this. The system service *'AccountManagerService'* allows apps to save account information with its public interface *'addAccountExplicitly()'*. However, as shown in the figure, inside the interface, the data storing operation *'db.insert()'* is exposed, which adds the input (e.g., *'account'*) into a database and saves the injected account information permanently. Although the system service validates if the account being stored belongs to the requester app, it does not limit the number of accounts that an app can store. Hence, a malicious app can store a huge number of accounts by requesting this service many times, which finally makes the size of the account database (also always loaded in memory) exceed the memory ceiling of the service (e.g., for Pixel 3XL, the heap memory is limited to 512MB). As a result, the attacker breaks down the system service and causes the entire system to reboot. Worse still, every time when the Android system is rebooted, the critical account system service is required to be started, and the account database is loaded into memory. Nevertheless, due to the too big size of the database, the service quickly occupies too large memory and is shut down. Consequently, the Android system falls into an endless loop of rebooting and crashing, which leads to a permanent DoS attack.

Furthermore, we discover a rich range of attack objects (i.e., exposed data storing instructions), not just the operations related to saving users' or apps' information (e.g., account). For example, the window system service *'WindowManagerService'*) creates a window session for capturing user's operations on the current screen. The window session is stored in a container and can also be attacked (see more details in §II-B).

As mentioned above, one (major) cause of the security flaws is no protection on data storing operations in system services. The life-cycle design of the containers (e.g., array, set, database, etc.) in Android system services is another important cause. In order to provide seamless user experiences, Android system services are always ready to serve apps. Specifically, the system services always prepare various containers for storing necessary data of an app, no matter the app has been opened or not. It is worth noting that this is a fundamental feature in Android. For example, Android allows apps to register many event listeners, which will be triggered when specific system events occur. The corresponding system service creates such an event listener container before an app is started and may release it after the app is closed. Therefore, there is a long time window between the time to store the app's event listener in this container (*Time-to-store*) and the time to

DOI 10.1109/SP46214.2022.00032

release it (*Time-to-release*), which gives attackers chances to launch attacks.

In this paper, we refer to this DoS attack as **Straw**[1] attacks. In essence, straw attacks belong to a kind of space DoS attacks, which rely on the *cumulative* effect on the target data storing (i.e., straw) – Each straw has little influence on the service memory, but the attacker can cumulatively increase the influence by continuously calling the vulnerable interface within the time window between *time-to-store* and *time-to-release*.

After understanding the straw attacks, we aim to create an automated vulnerability detection and verification tool against them. However, this is not an easy task. It should address several challenges and achieve the following goals:

- **Covering as many data storing instructions as possible.** The Android system will prepare plenty of resources before apps' requests. However, no detailed documentation can tell us how the corresponding mechanisms are designed and implemented. Besides, there are not well-defined subsystems and they are scattered across the huge code base of Android. For example, the documentation for the "audio" service only illustrates the interaction with audio hardware. However, when an app initializes an audio player, its audio configuration information is stored in the system service, while this configuration is only used when the registered audio player triggers specific events. Hence, the new tool should understand the data storing process and find as many types of data storing instructions as possible.
- **Supporting the testing of a large number of different types of service interfaces with different but proper inputs.** In Android, there are usually a large number of different service interfaces. These interfaces have different functionalities with very complex code logic. Understanding them needs strong domain knowledge. What is worse, different interfaces usually require different types of inputs. Hence, the new tool should be generic and scalable, and independent of domain knowledge. It is better to feed the service interfaces with proper inputs and guide the services under test to trigger as many vulnerabilities as possible.
- **Carefully monitoring attack effects in a lightweight but efficient way.** Launching straw attacks often needs cumulative effects. The result, i.e., memory change, of triggering an exposed data storing operation is often subtle and hard to monitor. Therefore, the new tool should be sensitive to any memory change. However, only memory change is inefficient to detect the vulnerability as it cannot indicate if the time window is enough for exhausting service memory. Thus, to confirm a vulnerability, we should trigger the attack effects, e.g., crashes of system services. Moreover, to trigger such a consequence, each data storing operation may need to be tested thousands of times. Hence, the new tool should conduct analysis in a lightweight way.

[1]This attack relies on continuously injecting data into Android system services, which looks like continuously adding straws on the back of a camel and finally breaking it down.

To accomplish these goals, we propose a directed grey-box fuzzing (DGF) based approach, named `StrawFuzzer`. Our basic idea is to continuously send input data (via an Android API call) to a server process and monitor its memory size change to see if it can release the injected data in time. The key design of `StrawFuzzer` is a novel combination of static and dynamic analysis techniques to effectively and efficiently detect and verify straw vulnerabilities in Android system services. First, we observe that static analysis often has the whole picture of all system services, and can somehow locate all potential data storing instructions. However, static analysis is difficult to get the runtime memory status. Hence, we introduce dynamic fuzzing as a complementary approach which can not only monitor the runtime memory usage but also generate a PoC to confirm the vulnerability.

Following the above methodology, `StrawFuzzer` is designed as a two-phase analysis tool. In the first phase (i.e., static analysis), `StrawFuzzer` first applies static program analysis techniques on system service code to generates call graphs, control flow graphs, and data flow graphs. Next, by leveraging the results, `StrawFuzzer` designs several heuristic rules to locate as many vulnerable data storing instructions as possible and use them as fuzzing targets in the next stage (i.e., dynamic fuzzing). Additionally, `StrawFuzzer` can extract the related path constraints to generate high-quality seeds. In the second analysis phase (i.e., dynamic fuzzing), `StrawFuzzer` intends to verify if the exposed data storing can exceed the memory limitation. First, `StrawFuzzer` uses an instrumented but lightweight environment to monitor any change of memory size carefully and calculate the seed's distance to the fuzzing target. Second, `StrawFuzzer` prioritizes seeds with an adaptive strategy by making a good trade-off between path exploration (to reach the target) and vulnerability exploitation (to exhaust memory). Last, when a vulnerability is confirmed, `StrawFuzzer` collects all results to generate exploits for the vulnerability verification purpose.

We evaluate `StrawFuzzer` on 3 popular Android systems with all the security updates (i.e., Android 10.0 on Pixel 3, Android 11.0 on Pixel3 XL, and Android 10.0 on Oneplus 7). As a result, `StrawFuzzer` successfully discovers 35 straw vulnerabilities as well as 474 vulnerable service interfaces (as indicated in Table II), which affect about 35% of Android system services. Even high-privileged interfaces can be attacked. We further analyze 3 customized Android systems from Huawei, Samsung, and Vivo, and find that they inherit most of the vulnerabilities from stock Android and all can be attacked. Furthermore, the attack speed can be controlled by the attacker. We find at least 42% of attacks can be finished within 1 second and 90% within 77.6 seconds. Additionally, we confirm that at least 3 vulnerabilities can also be used to attack services provided by Android apps since regular app services share the data storing vulnerabilities with the system services. Specifically, we collect the top 100 free apps from Google Play and confirm 76 of them suffer from straw attacks.

We have responsibly disclosed our findings to Google (in Oct 2020), Huawei (in Apr 2021) and Oneplus (in May 2021).

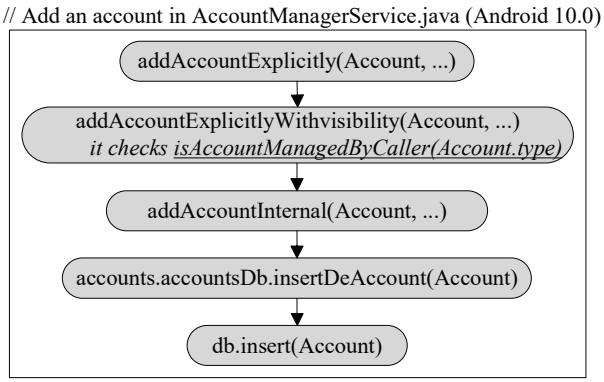


Figure 1: The simplified call chain of adding an account in AccountManagerService. It only checks if the account belongs to the caller.

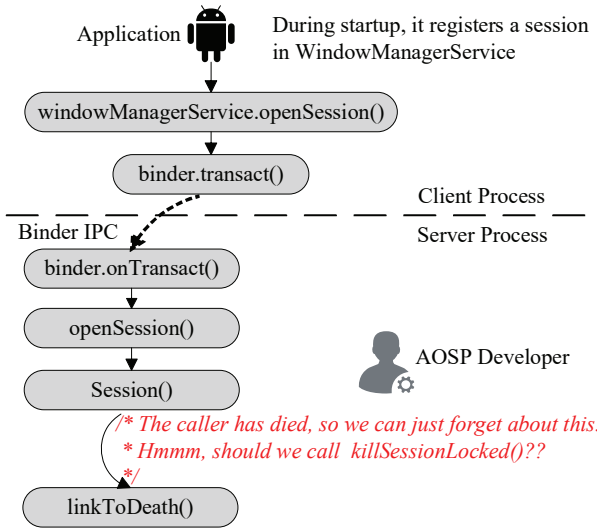


Figure 2: The simplified call chain of registering a window session in WindowManagerService.

They confirmed and acknowledged our findings, as well as decided to deploy patches to fix them.

We summarize the contributions of this work as follows:

- Our work is the first to systematically uncover the straw attacks in Android systems, as well as revealing the root causes of this new kind of vulnerability on Android services' design and implementation.
- We design and implement a novel tool, called `StrawFuzzer`, which can automatically locate and exploit the vulnerable interfaces in Android services by utilizing a DGF based approach. The source code is available at GitHub [1].
- We evaluate `StrawFuzzer` on 3 Android systems. The results show `StrawFuzzer` successfully discovers 35 straw vulnerabilities and exploits 474 vulnerable interfaces. Besides, we confirm popular Android apps also suffer from straw attacks.

II. Understanding Straw Attacks

In this section, we explain how straw attacks perform and why the vulnerabilities are universal in Android. For consistency in this paper, we use the term *server process* as a process that receives data, *client process* that sends out data in inter-process communication, and *public interface* as the Android API call provided by server process. Besides, the client process *injects* data into server process by sending requests containing input parameters. Since both Android system and apps can provide services for receiving data, both of them can act as server processes and are affected by the straw attacks.

A. Straw Attacks in Android Services

Straw attacks leverage the logic flaw in server code to exhaust its memory resources. That is, it does not limit the total amount of data that can be stored from client process. To conduct such an attack, the malicious app does not require a special configuration and just looks like a normal app. Specifically, during each iteration, the malicious app calls a public interface in Android services. Then the Android Binder will provide the needed inter-process communication (IPC) to connect the client and server process. Next, the target public interface is executed in server process with input data coming from client process. Starting from executing this interface, an execution path drives the control flow to arrive at the vulnerable data storing instruction. After the execution is finished, the data is stored and retained in the server memory. Note that the malicious app calls the interface normally and behaves just like all other benign apps. Thus, it's difficult to separate it from other apps. The only difference is that the malicious app will call the interface multiple times.

Android contains various system services (e.g., 200 services in Android 11.0) and apps (over 2,800,000 apps in Google Play). Though they implement different functionality based on Binder IPC, the underlying mechanism is the same – The client data is first serialized in *Parcel* and sent out by *binder.transact()*, then the server process deserializes the data from *binder.onTransact()* and propagates it to its local method. If the transformed data are stored in the server memory, they will consume the memory resources of server process. Thus, the straw attacks on them are similar. In addition, Android Binder uses a fixed-size buffer to prevent the transformed data from exceeding 1MB, and this limitation is shared by all transactions in progress [2].

B. Analyzing Root Causes for Straw Attacks

Inconsistent Life Cycle. By using Binder based IPC, client process can send data to server process. However, the life cycle of these data is inconsistent in client process and server process because of the time window between the data storing (*Time-to-store*) and the data releasing (*Time-to-release*). More specifically, when the IPC ends, the client process can free the data and recycle the memory occupied by them. But the server process will hold the data for a long time in case that it will be used in the future. For example, in Figure 2, the client app creates a window session for processing user operations. Thus,

```
// android.hardware.SystemSensorManager.java
1. private static final MAX_LISTENER_COUNT=128;
2. protected registerListenerImpl(…){
3.    … …
4.    if(mSensorListeners.size() > MAX_LISTENER_COUNT){
5.          throw new IllegalStateException( "register failed," + "the sensor
            listeners size has exceeded the maxium limit"
            + MAX_LISTENER_COUNT);
6.    }
7. }
```

Figure 3: The number of registered sensor listeners in Sensor service cannot exceed 128.

for each session, the server and client process both maintain several data structures to store this session's data. However, if the client app frees the memory of these data structures (i.e., through GC in Java), the server process will still hold this session. In fact, by calling *linkToDeath()* which binds this session with the client app's life, the server process will not recycle this session unless the client app is dead. Thus, the inconsistent life cycle of the data objects transformed through IPC will make the server process consume more memory resources than client process.

The Android system developers may also be confused about the life cycle of the data stored in system server. One example is their comments in Android's code illustrated in Figure 2. They think that once the caller is dead, this session object should be freed by *linkToDeath()* and they wonder if they should actively use *killSessionLocked()* to recycle it. However, based on our experiments, *linkToDeath()* will not kill the session object immediately when the caller is dead. Actually, it needs about 385 seconds on average if we shut down the caller app, and about 450 seconds on average if we uninstall the caller app. This time window can further be abused by attackers to mislead the mobile user. For example, the attacker app can consume most of the system server memory resources which are just a bit smaller than the upper bounds. After the user shuts down the attacker app and starts another app, the system server will crash once this app consumes server memory, which re-delegates the role of the attacker to another app from the perspective of mobile users.

Limited Memory Usage. Android enforces a set of limitations on each process's memory resources, including the system server, which indicates that the total amount of stored data in server memory has an upper bound. For example, Android limits the heap memory of each process to 512MB in Pixel 3XL (Android 11). If the system server exhausts its heap memory, it will throw out a 'java.lang.OutOfMemoryError' and crash. Besides, Android also customizes plenty of memory resource limitations. For instance, as illustrated in Figure 3, the number of registered sensor listeners in "sensor" service cannot exceed 128. Otherwise, it will throw a 'java.lang.IllegalStateException'.

The exhaustion of these limited memory resources will raise the attention of Android's recovery mechanisms, such as Watchdog and ANR [3]. Commonly, these mechanisms are designed to reboot the system if it runs into an error state occasionally. However, it's easy to see that, with the capability of exhausting system server memory resources, the attacker can control when and how often the system server will run into an error state, which is far beyond the design purpose of Android's recovery mechanisms. Besides, though Android implements plenty of system services, most of them run in the same process, i.e., the system server. Thus, one system service's crash can break down the whole system server as well as other system services.

Lack of Memory Size Check. A straightforward way to ensure the server process avoids memory resource exhaustion is checking the size of available memory before storing client data. However, we find that some memory size checks are incomplete. Figure 4 illustrates an example. The "window" service of Android expects each client app only registers one *Session* object to interact with the window manager. In fact, a malicious app can register as many *Session* objects as it wants because of lacking essential checks, which finally exhausts the server memory.

Besides, there exist design flaws in Android IPC, which pose challenges for servers to enforce memory size checks. For example, Figure 5 illustrates two interfaces provided by Android IPC for deserializing client data. Compared to the *createFloatArray()*, *createStringArray()* does not check if the server's available memory is enough for deserializing client's data. However, to enforce such a memory size check, the server needs to know the exact size of the memory that client's data will consume. For *createStringArray()*, the server should know the array length (i.e., N in line 2) and the memory size of each *java.lang.String* object. Unfortunately, the memory size of each *java.lang.String* object depends on the number of characters in it, which is undefined and can be any int value. Thus, the server process cannot compute the exact size consumed by the string array and compare it with the available memory. This flaw can be abused to attack plenty of public interfaces, even the privileged ones, illustrated in §VI-B.

Lastly, the enforced memory size checks are double-edged swords to all apps, which also can be abused to launch DoS attacks. For example, though the check of available memory (e.g., check dataAvail()) in *createFloatArray()* can prevent the server from memory exhaustion, it also prevents all the apps from sending data to the server process if the server does not have enough available memory. Thus, if a malicious app first stores a lot of data in the server process, which is a bit smaller than the memory size limitation, no other apps can use this server anymore.

III. Design Insights and Approach

A. Static Analysis

By utilizing static analysis, we can locate all data storing instructions (with heuristic rules) and service interfaces that can reach these instructions. Furthermore, we can collect path constraints to these data storing instructions and generate initial seeds for dynamic fuzzing.

```
/**
 * This class represents an active client session.  There is generally one
 * Session object per process that is interacting with the window manager.
 */
class Session extends IWindowSession.Stub ... {
   ...
}
```

Figure 4: The Android framework developer expects each client process has one session.

```
// android.os.Parcel
public final String[] createStringArray() {
    int N = readInt();
    if (N >= 0) {
        String[] val = new String[N];
        for (int i=0; i<N; i++) {
            val[i] = readString();
        }
        return val;
    } else {
        return null;
    }
}
```

No Check

```
public final float[] createFloatArray() {
    int N = readInt();
    // >>2 because stored floats are 4 bytes
    if (N >= 0 && N <= (dataAvail() >> 2)) {
        float[] val = new float[N];
        for (int i=0; i<N; i++) {
            val[i] = readFloat();
        }
        return val;
    } else {
        return null;
    }
}
```

Check dataAvail()

Figure 5: createStringArray() *vs.* createFloatArray()

1) Heuristic based vulnerability candidates locating: One major challenge is that no documentation discovers how many data storing instructions exist in Android, where they are, and what they look like. Hence, we need first to figure out the pattern of these data storing instructions and then identify them by static analysis.

Intuitively, the instruction *'new'* may give some hints, since it creates a new object and consumes memory resources in Android, which is a Java-based platform. However, these data objects will normally be freed (i.e., GC in Java) after the method's execution. Thus, they will not be "stored" in the server memory, which indicates that labeling all the data operation instructions will introduce a lot of false positives.

Instead, considering our goal is to abuse data storing to launch straw attacks, we observe that they should satisfy several constraints. First, the server should maintain references of the injected data after the end of execution to prevent them from being recycled by GC. That is, utilizing an object in server's main thread to keep their references. Second, the size of the data structure used for maintaining them can be increased, which requires that these data structures are certain containers that store references of injected data, e.g., *android.util.ArrayMap*. We detail this part in §IV-A.

2) Initial seed creation: The efficiency of fuzzing depends on the quality of the seeds. Traditionally, prior work [4]–[8] utilize expert experience to select initial seeds and generate new seeds by mutating them. However, with a huge code base and plenty of service interfaces, it's challenging to summarize enough expert knowledge for seeds used for fuzzing Android system services. Thus, we propose a novel approach by automatically extracting such knowledge from Android framework. The idea is that, in the Android framework, the path constraints (i.e., branch conditions) are highly relevant to system status or the system resources can be accessed by the caller app. These inputs have confined values for choice because of the limited run-time environment. For instance, the input *pid* of interface SliceManagerService.checkSlicePermission() can be dynamically determined by *android.os.Process.myPid()*. Therefore, we can extract these values from Android framework and use them as initial seeds.

Note that, during the control- and data-flow analysis, we also split the public interfaces' parameters into three sets: 1) control-flow related inputs, which taint values used in checking branch conditions, 2) consumption related inputs, which taint data injected through those labeled data storing instructions, and 3) other inputs, which are irrelevant to branch conditions and data storing. The idea is that `StrawFuzzer` only needs to mutate control-flow related inputs during exploration to find new paths and consumption related inputs during exploitation to trigger cumulative consequences.

B. Dynamic Fuzzing

Although the static analysis can help locate the interfaces that store input data, they cannot determine the attack effects. For instance, the attacker can add broadcast messages into a queue maintained by "activity" service. Meanwhile, the system server could process and remove them from the queue. The attack can succeed only if the speed of storing data is faster than that of releasing data. Thus, dynamic analysis is necessary for verifying if the time window is enough for exploiting the data storing.

1) Lightweight feedback collection: we first collect the feedback of exploration (i.e., the distance to the target) and exploitation (i.e., the memory size change) to guide fuzzing.

First, to calculate the distance to the target, prior work utilizes basic block level instrumentation to obtain the "distance" between execution traces and target sites [9]. However, this introduces non-negligible overhead during the run-time, which significantly affects the monitoring of memory size change. We first design and implement a lightweight but efficient approach. Considering that the invocations of data storing instructions rely on the satisfaction of preconditions in their caller methods' control-flow structures, we can obtain the execution traces at the method level and use the control-flow information as the guidance to improve them. Thus, we can calculate an approximate probability of reaching the target for each method on call graph (CG) under the guidance of control-flow graph (CFG), and use this reachability probability as the "distance", detailed in §IV-B.

Next, for monitoring memory size change, the key problem is that each straw (i.e., input data) has little influence on the memory size, even smaller than the regular change of server memory. Thus, we should execute each seed multiple times to accumulate its influence so that we can observe the memory size change. However, there exist some constraints that implicitly determine if the app-provided data will retain in server memory. For example, the system server may ignore the inputs with the same value. Thus, to evaluate a specific seed, we should generate a set of variations that have the same quality but different values. Specifically, we generate a new

input by modifying its content without changing its size, to ensure it has the same quality (i.e., consuming the same size memory) as the original seed.

2) Adaptive seed selection: In general, StrawFuzzer favors seeds that get closer to the target in exploration or consume more memory in exploitation. But a critical obstacle here is how to balance the trade-off between exploration and exploitation [9], (i.e., when to select seeds for exploration or exploitation). Traditional approaches like AFLGO assign fixed time to split them. However, such a splitting heavily relies on the empirical knowledge of the tested program and is inflexible. Since Android system services provide plenty of public interfaces, it's impractical to have enough empirical knowledge to pre-set the splitting time for each of them. Besides, the exploitation in StrawFuzzer can be time-consuming and it's hard to determine the exact time needed. Thus, to balance the trade-off, StrawFuzzer uses an adaptive strategy to dynamically schedule them. Specifically, StrawFuzzer introduces simulated annealing to balance the scores (i.e., energy) of each seed used in exploration and exploitation, and selects the seed with the highest score in each round of fuzzing, detailed in §IV-C.

3) Exploit generation: Since StrawFuzzer finally triggers the attack consequence (i.e., crashes of system server) for each vulnerability, one straightforward way to generate exploits is collecting the high-quality seeds generated by the fuzzing and re-using them to construct the corresponding exploits. However, a problem for exploiting such vulnerabilities in the wild is that it is undetermined how many times executing the vulnerable interfaces can exhaust the memory resources, which are affected by the run-time environment and device performance. Thus, the number of high-quality seeds needed to exploit the vulnerability is also undetermined, which requires that the exploits should have the capability to generate new seeds. The idea here is similar to the exploitation phase in dynamic fuzzing – We can use these collected seeds as high-quality initial seeds and arm the exploits with the capability to generate new seeds by mutating them.

IV. Methodology

This section details our methodology. Figure 6 illustrates the overall architecture of StrawFuzzer and the Algorithm 1 in the Appendix shows the high-level design of the fuzzing process.

A. Static Analysis

The first step of StrawFuzzer is to locate the vulnerability candidates and generate initial seeds for testing them.

1) Labeling data storing instructions: Based on the aforementioned heuristics, we first identify all the fields defined in Android system services, as they all run in the system server process in singleton mode. Note that, since global static objects are also singleton in server's main thread and will not be freed by GC, we additionally identify all the static fields in the Android framework. Then we pick out containers in them including various arrays, sets, maps, queues, lists, and databases. Next, we label all the operations that increase the size of these selected containers like *add(), put()*, and *insert()*. As a result, we find 96 kinds of instructions in Android and label them as fuzzing targets.

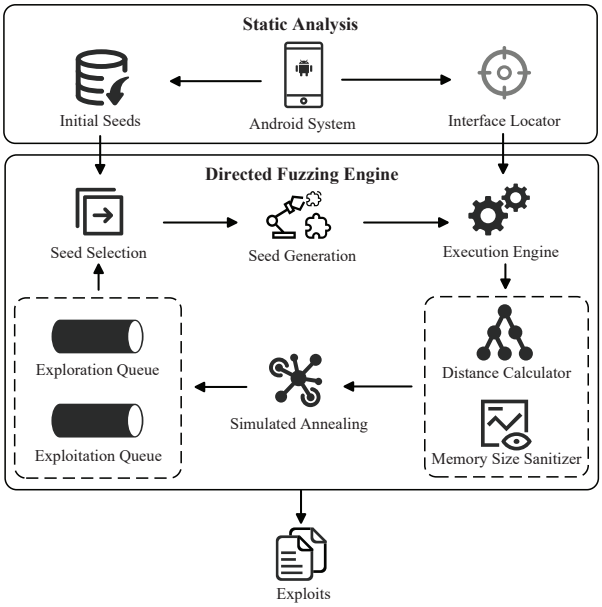


Figure 6: The overall architecture of StrawFuzzer.

```
// ActivityManagerService.java
1. public int broadcastIntent(⋯){
2.   ⋯ ⋯
3.   if(userId!=UserHandle.USER_ALL)  ⋯ ⋯ ;
4.   if(userId==UserHandle.USER_CURRENT_OR_SELF) ⋯ ⋯ ;
5.   if(userId==UserHandle.getUserId(⋯) ⋯ ⋯ ;
6. }
```

Figure 7: An example for extracting initial seeds from Android.

2) Labeling public interfaces: To locate the public interfaces which can reach these fuzzing targets, we conduct a backward control-flow analysis from them, as well as a data-flow analysis to check if they are tainted by the public interfaces' parameters. As a result, we match 609 fuzzing targets with 1,244 public interfaces and use them to conduct dynamic fuzzing.

3) Initial seed creation: StrawFuzzer selects initial seeds by extracting all branch conditions checked in the Android framework. Commonly, they check the input by comparing it with specific constants or dynamic system status (e.g., *UserHandler.getUserId()*). Thus, we directly use these constants and the return values of these functions as initial seeds. An example is illustrated in Figure 7. StrawFuzzer will add *UserHandler.USER_ALL*, *UserHandler.USER_CURRENT_OR_SELF* and the return value of *UserHandler.getUserId()* into initial seeds.

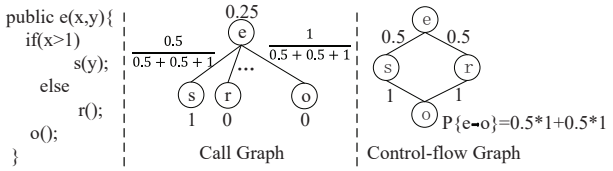


Figure 8: An example for computing the reachability probability from entry e to target data storing s on call graph with the guidance information collected from control-flow graph.

B. Feedback Collection

We detail the contribution evaluation of each seed in exploration and exploitation for guiding the dynamic fuzzing.

Calculating Distance. Figure 8 illustrates an example of this. Initially, we assign the fuzzing target s on CG with $P_r(s) = 1.0$, and nodes which cannot reach s with $P_r = 0.0$. Then we conduct a backward traverse on CG from s to compute the P_r of the rest nodes. Note that, during the traversing, we assign each edge in CG with their reachability probability computed based on CFG. For example, the probability $P_r\{e \rightarrow s\}$ from e to s on CFG is 0.5, because e has two successors including s in the CFG. Similarly, $P_r\{e \rightarrow r\}$ is 0.5 and $P_r\{e \rightarrow o\}$ is 1. Next, we assign these reachability probabilities on corresponding edges on CG. For instance, the probability $P_r\{e \rightarrow s\}$ from e to s on CG is set to 0.5. Then we normalize all the edges' probability to ensure none of the nodes' P_r exceeds 1.0 by dividing the total probabilities of all its brother edges. After normalization, $P_r\{e \rightarrow r\}$ is re-set to 0.5/(0.5+0.5+1), which is 0.25. Additionally, the probability P_r for each node is a combination of all its successors. For node e in CG, $P_r(e)$ equals $P_r\{e \rightarrow s\} * P_r(s) + P_r\{e \rightarrow r\} * P_r(r) + P_r\{e \rightarrow o\} * P_r(o)$, which is 0.25. Since P_r reveals its distance to the fuzzing target, we can use it to evaluate the contribution of seeds in the exploration phase. That is, for each seed, we can obtain the set of triggered methods S_m through instrumentation. Finally, for each method m in S_m, we calculate its $P_r(m)$, and select the max of them as the seed's contribution:

$$c_r = max\{P_r(m) | m \in S_m\}$$

Thus, if c_r equals 1.0, we can know the distance of seed to the target is 0. Algorithm 2 and 3 in the Appendix show the detail of computing P_r.

Sanitizing Memory Size. In order to observe the cumulative consequences of multiple executions, we design a memory size sanitizer to monitor the memory resource consumption during exploitation. Specifically, we choose two kinds of indicators for sanitizing memory size: (1) the heap size, which is the total amount (512MB in usual) of memory limited by Android JVM; and (2) customized memory sizes, which are scattered in Android system services based on their functionality. As the heap size can be monitored by specific system interfaces like *getMemInfo()*, the problem here is how to identify customized memory sizes and their upper bounds. Fortunately, with the interface locator, `StrawFuzzer` identifies a set of singleton containers in the system server. The next step is to identify whether Android enforces memory size checks on them and their upper bounds. Particularly, we conduct a control-flow analysis to locate the *if* statements that compare their sizes with a constant, and use the constant as the upper bound. As the example illustrated in Figure 3, the *MAX_LISTENER_COUNT* (i.e., 128) is identified as the upper bound of *mSensorListeners*.

With memory size sanitizer, `StrawFuzzer` can capture the memory difference before and after executing the public interface multiple times. Suppose that, before the execution, the server has an initial memory size m_i, and m_e after the execution. The total amount of the memory resource is m_t. Thus, the contribution to memory consumption can be calculated as:

$$c_m = \frac{m_e - m_i}{m_t - m_i}$$

Note that, as our memory size sanitizer monitors a set of memory resources in the target process including heap memory and customized memory resources, `StrawFuzzer` will evaluate the seed's contribution to them separately, and select the max one as the result.

C. Seed Selection

This step selects high-quality seeds to fuzz Android system services. Generally, `StrawFuzzer` favors seeds that can get closer to the targets or consume more memory resources, and uses simulated annealing to adaptively select seeds for exploration or exploitation, to prevent `StrawFuzzer` from wasting too much time on finding new paths or exploiting hard-to-exploit paths.

Simulated Annealing. The path reachability c_r and memory consumption c_m of a seed reveal its contribution to the fuzzing. However, seeds with high contribution do not always exploit the targets in practice due to some hard-to-satisfy path constraints or hard-to-exploit targets. In order to avoid `StrawFuzzer` getting trapped in a local optimum and adaptively schedule exploration and exploitation, `StrawFuzzer` uses simulated annealing to gradually obsolete inefficient seeds as follows:

$$T = 20^{-\frac{N}{500}}$$

in which N stands for how many times the seed has been selected. Thus, the final contribution c of a specific seed is:

$$c = (c_r + c_m) * T$$

By applying such a method, the seeds are evaluated by both their contribution and selection times. On the one hand, a seed in exploitation (i.e., $c_r = 1.0 \wedge c_m \geq 0$) is prioritized and has more chances to be selected than other seeds (e.g., $c_r < 1.0 \wedge c_m = 0$). On the other hand, if the seed has been selected multiple times and still cannot trigger the cumulative consequence, its final contribution will decrease

and StrawFuzzer will prioritize other seeds. Based on its final contribution, the energy of a seed can be evaluated as:

$$energy = \lfloor k * c \rfloor + b$$

in which b is the initial energy (i.e., 1 in our experiments), and k is 100 based on AFL [10].

Note that one data storing instruction in Android system may be reached from multiple public interfaces (e.g., *mDisplays.put()* is reachable from 60 service interfaces). Fuzzing these interfaces one by one will waste plenty of testing resources. Thus, during seed selection, StrawFuzzer will consider all the seeds and their interfaces that can reach the targets. That is, StrawFuzzer will map each seed to its corresponding interface. After selecting the seed with the highest energy, StrawFuzzer will use this seed's interface as the entry to conduct dynamic fuzzing.

D. Seed Generation & Mutation

1) Exploration Phase: As the goal of exploration is to find a seed that could reach the fuzzing target, StrawFuzzer mainly focuses on the control-flow related inputs of the tested interface in this stage. After selecting a seed for exploration, StrawFuzzer will generate a set of child seeds near it based on its energy by only mutating the control-flow related inputs. As an example, the interface *broadcastIntent* of Activity service has 13 parameters, where StrawFuzzer only focuses on mutating 8 control-flow related parameters to generate seeds in the exploration phase.

2) Exploitation Phase: To make sure the fuzzing target can be executed every time during exploitation, StrawFuzzer will not modify the control-flow related inputs in this phase, and only mutate consumption related inputs, because repeatedly invoking a service interface with the same inputs may not result in cumulative consequences. For instance, the public interface *isTagEnabled()* of Dropbox service in Android 10 takes a *String* as input, and uses it as the key for storing data into a static *ArrayMap* object. So if *isTagEnabled()* is called multiple times with the same input, only the first execution could consume memory resources in server process. Thus, mutation is inevitable in this phase.

In general, we mutate these consumption related inputs towards a trend of enlarging memory size. Specifically, we observe that these inputs are commonly used in two types: (1) the size of memory allocation, e.g., the size of arrays or maps; (2) the content stored in memory, e.g., the data stored in arrays or maps. For the first type, they are usually some specific *Integer* values. Thus, StrawFuzzer tends to generate a very large value for *Integer* and mutate it by generating new inputs near it. For the second type, the memory consumed by them commonly depends on the *String* objects stored in them, as the size of *String* objects in Java is unlimited. Thus, StrawFuzzer tends to generate different and large *String* values for them. For instance, the interface *accountAuthenticated()* has only one parameter of *android.accounts.Account*, which can be constructed with three *String* parameters named *name*, *type*, and *accessId* respectively. The first parameter *name* is identified as consumption related input while the last two parameters are control-flow related inputs, so StrawFuzzer will only mutate *name* by increasing its string length during exploitation.

E. Exploit Generation

To generate exploits, we first collect high-quality seeds during fuzzing, then inherit the seed mutation used in the exploitation phase to arm our exploits with the capability to generate new seeds. Note that, since the new seeds are used for exploitation, we only mutate consumption related inputs. Moreover, since our attacks are launched by third-party apps, we prepare a code template for sending requests to Android services through Binder IPC. Interested readers can find it in the open-sourced code of StrawFuzzer.

V. Evaluation

We implement StrawFuzzer with about 13,000 lines of Java code and 1,200 lines of Python code. Specifically, we use Smali/BakSmali [11] and vDexExtractor [12] to disassemble the Java bytecode of Android system, use Soot [13] to implement static analysis, and use Xposed [14] to implement method-level instrumentation for dynamic fuzzing. Then we evaluate StrawFuzzer's effectiveness, efficiency, and accuracy by applying it to 3 latest Android systems. We further analyze how the vulnerabilities affect real-world Android system and present some case studies in §VI.

Statistics of analysis target. Table I shows the tested system and overall results. Note that we update all systems with their latest security updates before our evaluation. On average, each system has 193 services and 4,097 public interfaces. Besides, after customization, the third-party vendor (i.e., Oneplus) adds more services into the system.

Efficiency. To illustrate the efficiency of StrawFuzzer, we summarize StrawFuzzer's analysis time on all the analysis targets, i.e., 3 Android systems with 579 system services. For a specific Android system, StrawFuzzer needs about 204 hours to finish its analysis on average. In particular, the interface locator needs about 4 hours and locates 435 candidates of vulnerable data storing instructions as well as 963 public interfaces. The dynamic fuzzing needs about 200 hours for each system, consisting of 148 hours for exploration and 52 hours for exploitation. When fuzzing each interface, the timeout is set as 300 seconds, experimentally.

A. Tool Accuracy

In total, StrawFuzzer outputs 673 crashes as well as 673 exploits after removing duplicates. Next, we break down these crashes and find that they are triggered by exploiting 40 vulnerabilities, of which 35 belong to *Unlimited Data Storing*, and 5 belong to *Uncaught Exception* [15] as by-products. Note that some interfaces can trigger multiple vulnerabilities. We get 474 unique interfaces after deduplicating the interfaces affected by all straw vulnerabilities. The details are shown in Table IV. The reason for by-products is that they can trigger the crash of system server, which is an indicator of

Table I: Impact of discovered straw vulnerabilities over popular Android system.

Android System (Version/Device)	# Vulnerable Service (# All Service)	# Exploitable Interface (# All Public Interface)	# Generated Exploit	Date of Security Updates
11.0 (Pixel 3XL)	70(200)	435(4,146)	598	2020.09.05
10.0 (Pixel 3)	67(183)	395(3,903)	579	2020.08.05
10.0 (Oneplus 7)	67(196)	400(4,243)	584	2020.07.01

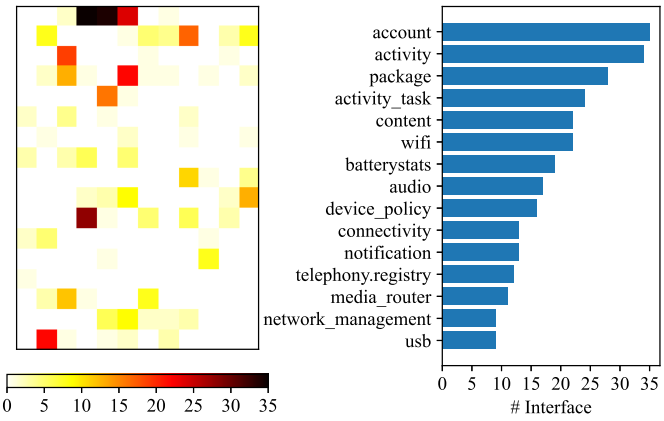
our memory size sanitizer. As `StrawFuzzer` successfully generates exploits for each vulnerability, the final detection precision is 100% and there are no false positives in the results.

For false negatives, since `StrawFuzzer` is designed to discover a new type of vulnerability (i.e., straw vulnerability), and there is no ground truth of all vulnerabilities, which is a common limitation for existing work [16]–[19] devoted to uncovering new types of vulnerabilities, we currently do not have a good way to predict the false negatives accurately. However, to understand the reasons which may lead to false negatives, we further analyze 50 randomly selected candidates that could not be successfully exploited. Overall, 43 of them fail to explore due to the path constraints, and the other 7 are hard to exploit in practice. The details are as follows. First, 34 of them fail because the corresponding service interfaces require privileged permissions only granted to system apps. Second, 2 candidates require critical parameters which are only allowed to be instantiated by system processes. For example, the method *grantDevicePermission()* of usb service has a parameter *android.hardware.usb.UsbDevice*, which will be checked whether its creator is system process before executing this interface. Third, 6 candidates require specific process or system status which are hard to satisfy, for instance, the interface *setBackupEnabled()* requires the current user has been registered in the backup service, which cannot be satisfied by `StrawFuzzer` via mutating inputs. Forth, there is one candidate whose execution triggers another crash before finishing the exploitation (i.e., uncaught exception). Lastly, 7 candidates are hard to exploit in practice, as the service interfaces limit the size of input data, and the exploitation cannot finish within the time limit. For example, `StrawFuzzer` could only inject an integer once into the interface *checkPackage()*, and needs more than 24 hours to exploit it successfully, which is far beyond the timeout, i.e., 5 minutes in our experiments.

B. Tool Effectiveness & Findings

For all found vulnerabilities, we illustrate their details in Table III (two interfaces labeled with $\star$ can be exploited for permanent DoS attack and the rest for temporary DoS attack) and list all exploitable interfaces in Table IV in the Appendix. We choose a subset of them as case studies in §VI to illustrate the attack in the real world. We further conduct some analysis to understand the root causes of these vulnerabilities and how they affect real-world Android systems.

Finding 1: Plenty of Android system services are exploitable. Figure 9 visualizes the number of affected interfaces per system service in Android 11. It shows that, among the 200 system services, 35% of them contain exploitable



(a) Distribution of exploitable interfaces in system services.

(b) Top 15 services containing the most exploitable interfaces.

Figure 9: Exploitable interfaces per system service in Android 11. Each node in (a) stands for one service (200 in total).

interfaces. Moreover, the most affected interfaces are from a small set of services containing account, content, activity, device policy, etc., which are correlated with device or app status and user information management. Besides, though the interfaces used for attacks are of various types and come from various services, the main influences on memory consumption come from parameters of type "java.lang.String". Interestingly, these strings commonly have no limitations, even though some of them carry distinct semantics like calling package names.

Finding 2: High-privileged interfaces can also be attacked. We then analyze the permission requirements of the 474 exploitable interfaces. 262 of them require no permission, and 54 of them require app-level permission. The rest (i.e., 158) are high-privileged interfaces, which are protected by system-level permissions. This is because the vulnerabilities lie in the deserialization of input data, which are executed before the permission check. Thus, attackers can finish the execution of vulnerable code (i.e., storing data in server memory) before failing the permission check. That is, to attack these privileged interfaces, the malicious app does not need any permission. We use a case study to illustrate this in §VI-B1.

Finding 3: Android deserialization significantly expands the influence scope of straw vulnerabilities. We further analyze the location of vulnerable data storing. As illustrated in Table II, though most of them (about 54%) exist in the particular APIs implemented in different Android services, they only affect a small part of exploitable interfaces (about

Table II: The summary of Straw vulnerabilities exposed by StrawFuzzer.

Location Category	# Vulnerable Data Storing	# Affected Exploitable Interface
Particular Service API	19	29
Android Deserializer	16	435
Total	35	474

6%). That is, over 90% of interfaces are exploitable because of the flaws in their deserialization phase. The reason is that Android provides a new deserialization mechanism, named *Parcel & Parcelable*, and all the Android service interfaces use the provided APIs to deserialize client inputs. Thus, if one of these APIs is vulnerable, all the service interfaces using it will be exploitable.

Finding 4: System updating of Android introduces new vulnerabilities. To reveal when these vulnerabilities appear in Android, we further conduct a cross-version analysis from Android 8 to 11. Specifically, for the 435 exploitable interfaces we found in Android 11, we analyze when the interfaces are added into Android (i.e., the *added_time*) and when they become exploitable (i.e., the *exploitable_time*). It shows that for about 6% of the interfaces, their exploitable_time is later than added_time, which indicates that these vulnerabilities are introduced by functional updates. For example, the vulnerability in *GnssManagerService.addGnssNavigationMessageListener()* can only be exploited on Android 11 because of a flaw in the update code. Specifically, Google updates 3 public interfaces in Android 11 to additionally accept a string type parameter *featureId*. Unfortunately, this string parameter will be stored in a set *gnssDataListeners* maintained by "location" service. Thus, a malicious app can abuse this to inject a lot of crafted strings into the system server.

Finding 5: Customized versions of Android inherit most of the vulnerabilities in stock Android. To understand how straw vulnerabilities affect customized versions of Android, we additionally collect 3 Android devices from Huawei (Android 10.0), Samsung (Android 10.0), and Vivo (Android 9.0). Note that these vendors heavily customize Android system and forbid users to root their devices. We cannot re-run StrawFuzzer on these systems as it requires system-level instrumentation. Thus, we choose to evaluate the exploits generated by StrawFuzzer on these devices directly. As a result, almost all vulnerabilities can be exploited successfully, except for one in the interface *trackPlayer()* of the "audio" service. This indicates that the customized versions of Android inherit most of the vulnerabilities from stock Android. The reason for the failed one is that Huawei modifies this interface's parameters by adding a string to indicate caller's package name. Additionally, we also identify a new vulnerability in one customized system service (i.e., the "secrecy" service) on Oneplus 7 (Android 10), which allows clients to register callbacks without limitation.

Finding 6: The time needed for the attack can be controlled by the attacker. We further analyze the straw vulnerabilities about the time it takes to render the device unusable. To reduce randomness, we test each vulnerability 3 times. As shown in Figure 10, 90% of straw vulnerabilities can be successfully attacked within 78 seconds on average. This time can be further reduced by leveraging parallel execution.

Note that, though straw attacks take a period of time to complete the attack, the attacker can interrupt and continue the attack at any time. Straw attacks exploit the vulnerable data storing in system server, which has a time window between the data storing (i.e.,*Time_to_store*) and the data releasing (i.e., *Time_to_release*). Thus, during this time window, the attack can be paused and the injected data (i.e., payloads) still exist in the server memory. Taking advantage of this feature, the attacker can control the time needed for the attack. On the one hand, the attacker can split the time needed into many small pieces and hide them into common tasks like file downloading. The user is hard to feel the attack because each piece only has a little influence on the system. We confirm that the time window is long enough for 95% of our vulnerabilities to conduct this kind of attack. On the other hand, the attacker can roughly inject plenty of payloads (like fork bomb and SYN flooding) into system server to exhaust its memory resources. In this case, the system will become unusable soon. For example, 42% of our vulnerabilities need less than 1 second to trigger the rebooting of system.

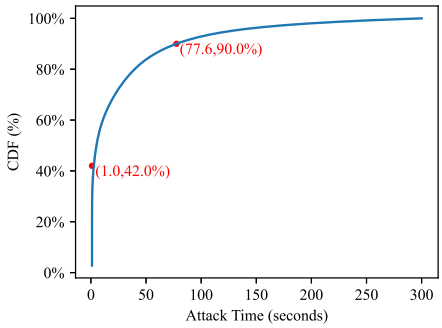


Figure 10: Attack time of straw vulnerabilities.

Finding 7: Android apps share the same vulnerabilities with Android system services. Like system services, the regular services in Android apps also need to store the apps' own data. In theory, if their internal data storing operations are exposed, attackers can launch straw attacks on these app services. To confirm this, we collect the top 100 free apps from Google Play, and then apply our exploit code to vet them. As a result, we find 76 apps suffer from straw vulnerabilities. We further break down the reasons for these vulnerabilities. On the one hand, they inherit the vulnerabilities in Android deserialization. For example, the popular "AppFlyer" SDK (70 billion+ installations) uses the vulnerable interface *createStringArray()* in Figure 5 to accept data from clients. Thus, any app (e.g., facebook) that uses this SDK is vulnerable. On the other hand, they implement vulnerable containers to store clients' data but without any limitation and protection, like using a *HashMap* to store the widget ids from other apps, which are used to bind specific calendar tasks.

C. Performance of Fuzzing

During fuzzing of these vulnerabilities, `StrawFuzzer` needs 28,746 executions of public interfaces on average to exploit the target. In order to figure out the test resources spent in each phase of fuzzing, we separate executions during the exploration and exploitation. The results show that `StrawFuzzer` performs more executions in the exploitation phase. As an example, the vulnerability *Account.Account()* lies in the deserialization of input data in IPC and can be quickly reached. Then `StrawFuzzer` will soon enter the exploitation phase. Thus, the strategy of assigning fixed time to two phases in prior work [20], [21] may need more time to trigger such a vulnerability. Furthermore, we analyze the switching times of scheduling these two stages during fuzzing and find that most of them are under 3, which also implies the effectiveness of our adaptive strategy to make sure each reachable path found has enough time to be tested in exploitation.

Comparison of adaptive strategy with fixed-time strategy. To evaluate the effectiveness of our adaptive strategy, we compare it with fixed-time strategy deployed by classical fuzzers like AFLGO [20]. Specifically, we randomly select 30 exploitable interfaces from our results and separately run `StrawFuzzer` with adaptive strategy and two fixed exploration time strategies – The first one divides exploration time and exploitation time as 5:1 which is the same with AFLGO, and the second 1:1. In this experiment, we set a timeout of 300 seconds and run each fuzzing setting on each interface for 5 rounds to reduce randomness. The result is shown in Figure 11, which illustrates the average crashes triggered per minute with different strategies. The result shows that our adaptive strategy finds vulnerable interfaces faster (about 4x) than the fixed-time strategy. Additionally, only `StrawFuzzer` successfully identifies all 30 vulnerable interfaces, while fixed-time strategy fails to discover 2 of them before exceeding the timeout. This is because adaptive strategy leaves more time for exploitation, thus accelerates vulnerability detection.

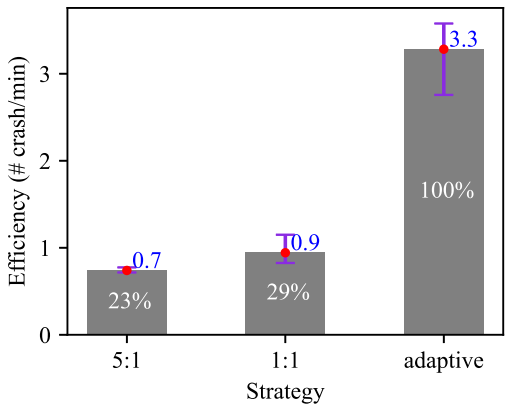


Figure 11: Efficiency of adaptive and fixed-time strategy.

VI. Case Study

We now choose a subset of our results to explain how they can be exploited.

A. Permanent DoS Attack to the Device

AccountManagerService provides an interface named *addAccountExplicitly()* that allows apps to store user accounts in Android system. Thus, apps can directly use these accounts to log in and reduce the user's effort to re-input the accounts. The stored accounts can be viewed in the *Settings* app of Android. Normally, each app only stores a few accounts in the system and mobile users can manipulate them (e.g., delete accounts) in the *Settings* app. Since account management is essential and widely used in Android apps, Android removed the permission requirements for this interface from Android 6.0.

1) Unlimited Registration of New Accounts: The interface does not limit the number of accounts that each app can add. Thus, a malicious app can store a large number of crafted accounts through this interface. AccountManagerService will store the inputted accounts to a database in the device's internal storage, i.e. */data/system_de/0/accounts_de.db*. During the startup of Android system, the system server will automatically load all the accounts stored in this database, which exceeds the memory limitation, and the system crashes again. The Android system will fall into an endless loop – it continuously reboots and crashes, which is a permanent DoS attack on the device.

The only way for mobile users to remove these crafted accounts is to use the *Settings* app. However, this app relies on AccountManagerService to manage these accounts. Since AccountManagerService is down, the *Settings* app will be stuck at the startup or crash directly. Worse still, other countermeasures that rely on *Settings* app like factory reset and system image flush will also lose their effects.

2) Exploitation in Real World: This vulnerability can be easily exploited in the wild. As aforementioned, this interface does not require any permission. To generate crafted accounts, malicious apps need to generate two strings – account name and account type. Since account type is pre-defined in each app, malicious apps can use a long string as the account name to construct a large account object. Figure 12 illustrates a simplified exploit. Note that, AccountManagerService ignores accounts with the same name and type, thus the account name should be different in each round of attack.

```
// Simplified Exploit (Tested on Pixel 3XL, Android 11.0)
while(true){
    AccountManager accountManager =
                    AccountManager.get(getApplicationContext());
    String name = getRandomAndLongString();
                    // Generate a random and very long string
    String type = "badAccounts";
    Account account = new Account(name, type);
    accountManager.addAccountExplicitly(account,"",new Bundle());
}
```

Figure 12: The simplified exploit for exploiting addAccountExplicitly() to conduct permanent DoS attack.

B. Bypass User Interaction to Reboot Android System

To protect Android users, Google implements a set of user interactions before triggering some critical behaviors of

Android, for example, rebooting the phone. We now give two examples to illustrate how to use these vulnerabilities to reboot the mobile phone without user interaction.

1) Deserialization Error of Privileged Interfaces: AccountManagerService provides a public interface *getAccountsAsUserForPackage()* used for querying the accounts of another user. This interface is protected by a system-level permission *INTERACT_ACROSS_USERS_FULL*, which cannot be granted to third-party apps. Thus, when fuzzing this interface, `StrawFuzzer` often triggers a security exception and is rejected from accessing this interface. However, `StrawFuzzer` still finds that the server memory consumption increases rapidly through calling this interface.

Specifically, this interface receives *Account* as input. Based on the design of Android IPC, system services will first deserialize the client's input data and then execute the service's code, including checking permissions. Thus, the code used for deserializing *Account* is not protected, and any vulnerability in it can be attacked by malicious apps. During the deserialization of *Account*, it will store itself to a global static set (i.e., *sAccessedAccounts*). If the malicious app continuously calls this vulnerable interface with different *Account* objects, the server memory consumption will increase rapidly and it finally triggers *java.lang.OutOfMemoryError* which crashes the system server and reboots the mobile phone.

2) Unlimited Increase of Cache: Android provides a system service, named *DropboxManagerService*, to record chunks of data from various sources, such as app crash logs and kernel error logs. For convenience, it labels the data with a *tag* before storing it in this service. Besides, it provides a public interface *isTagEnabled()* with a string parameter *tag*, which can be used by Android apps to check if the *tag* is allowed in this system.

However, `StrawFuzzer` finds that the system server memory will increase rapidly if an Android app continuously calls *isTagEnabled()* with different input *tags*. The reason is that this interface uses a hash map *mValues* to cache each tag's check result. Specifically, the interface *isTagEnabled()* should query a local database to ensure if the input tag is allowed to use. To accelerate the check and reduce queries to the database, it uses the hash map *mValues* to store the check results. Thus, for each tag, it first looks for the result in the cache. If the cache misses, it queries the local database and then stores the result in the cache. Unfortunately, it does not restrict the maximum size of the cache. Thus, a malicious app can increase this cache by using different *tags* and finally exhausts system server memory. Then the system server will shut down and the system will reboot.

VII. Discussion

Straw Attacks in More Scenarios. As storing data from the client is common in multi-task systems, we believe that straw attacks can be launched in more systems. However, the attack effect is determined by the limits set for memory resources of server processes. For servers with sufficient resources, the difficulty of straw attacks will be greatly increased, for example, Linux kernel, which can have a large amount of memory by using swap space. Besides, the recovery mechanism for programs running into an error state can determine the attack consequence. Nevertheless, straw attacks can indeed be launched against a server with limited resources.

Security Implications of Straw Attacks. Typically, straw vulnerabilities can be used to launch various DoS attacks.

1) Temporary DoS attack: Existing Android DoS work [16], [17] studied the security hazards that temporary DoS attacks may cause. With the help of a UI state inference attack [22], attackers can trigger the vulnerabilities when the system is conducting critical tasks. For instance, a malicious app could hinder the critical application patching by registering one receiver with the *package_removed* action and offline reverse-engineering. Android OS is widely used in various mission-critical scenarios like serving medical devices [23] and aircraft navigation [24], as well as embedded in nano-satellites [25]. Prior work [26], [27] illustrated how to infer the critical moments via side channels. Thus, a stealthy attacker can launch straw attacks at some critical moments when the above apps are running.

2) Permanent DoS attack: Although straw attacks do not directly destroy data, they bring great difficulties for victims to access data (e.g., messages and images). With a permanent DoS attack, attackers can cut off the interactions between users and their devices by trapping the victim device into an endless loop. Moreover, nowadays, many IoT, web, and cloud services, like Google accounts, are allowed to be strongly bound to smartphones to provide a better and more secure user experience. Normally, when a user accesses the service (especially in a new environment), the user's smartphone will receive an "approve-or-deny" message for authorization and verification. However, when the victim device is permanently down, the user faces trouble accessing the service, which may be totally locked.

3) Attack Re-delegation: Another feature of straw attacks is that attackers can re-delegate the attacker role to another benign app. One way is to crash the system when users using victim apps – We confirm that the malicious code can execute in the background in the latest Android (e.g., Android 11). Another way is consuming server memory to a bit smaller than the limitation (the upper bound). Thus, the next opened app can easily exceed the limitation, detailed in §II-B.

Possible Mitigation. A straightforward way for mitigation is to limit the number of memory resources, e.g., accounts, to a reasonably low number. This can be leveraged to protect known memory resources, but the obstacle here is that system defenders can hardly know all the memory resources that should be well limited, as data storing operations are widespread in Android system and involves a variety of memory objects. Additionally, the reasonable number is sometimes hard to decide, especially in the deserialization phase. For example, as indicated in Figure 5, the server process can not consume the exact size of memory consumed by the string array in a single request. Based on the understanding of the root causes, we propose some recommendations to

mitigate the straw threats. First, considering that Android deserializers affect the most vulnerabilities, we recommend that Android should enforce more checks and validation on data storing operations. The second mitigation is to shrink the attack window (between time-to-store and time-to-release) by enhancing services' capability of releasing data in their containers. For example, when the available memory is not enough, it can actively release the low-priority data (e.g., data from the same caller) with a first-in-first-out strategy. Last, when abnormal memory consumption is observed, further data storing operations should be warned and even paused to provide system services time to apply the above two mitigation strategies.

VIII. Related Work

Memory consumption vulnerability. Memory consumption vulnerabilities [28] commonly lead to the exhaustion of limited resources. Wang et al. [29] and Carbonneaux et al. [30] focused on worst-case memory consumption analysis to determine the bounds for programs. Numerous related studies [31]–[34] were devoted to detecting memory leaks in programs with the help of AddressSanitizer [35] and LeakSanitizer [36]. These studies detected memory leaks by locating unneeded but unreleased data in program memory, which was not suitable for Android. This is mainly because the input data stored in server memory may be used in the future thus could not be considered as unneeded objects, e.g., various event listeners.

Android DoS attack. Prior work [16]–[18], [37]–[39] studied DoS attacks to Android system. For instance, Huang et al. [17] discovered a design trait in the concurrency control mechanism of Android system server and Armando et al. [37] abused the loosely protected Unix socket permission to fork an unbounded number of processes. Similar to typical DoS attacks like fork bomb [40] and SYN flooding [41], these attacks commonly focused on specific limited resources, such as synchronized locks or processes. However, straw attacks can attack many different types of objects (e.g., account databases and data listeners). Furthermore, benefited from the diversity of data stored by system server, straw attacks can achieve high severity attack consequences, e.g., permanently downing the victim mobile device.

Fuzzing based vulnerability detection. Fuzzing is an effective technique that has been widely adopted to detect software vulnerabilities and bugs. However, state-of-the-art tools [4]–[8], [42]–[49] hardly satisfy the requirements of detecting straw vulnerabilities. First, existing work is not designed for testing exposed data storing operations. It is difficult to extend them to understand the data storing process inside a large number of Android system service interfaces and even to achieve the goal of covering as many different types of data storing operations as possible. Second, most fuzzing tools (e.g., [4]–[8]) heavily relied on domain knowledge for the purpose of optimizing the seed generation or scheduling. For example, AFLGO [20] used expert experiences to split the time used for exploration and exploitation. However, they are not suitable to analyze a large number of Android system services, as different Android system services provide completely different functionalities. Hence, a generic solution regardless of domain knowledge is needed. Some work proposed mitigation to this problem, but they are still ineffective to detect straw vulnerabilities. For example, Vuzzer [50] used static analysis to aggressively extract data-flow features (i.e., hard-coded constants) from condition checks to assist input generation for coverage based fuzzing. However, we should first understand the data storing process and then identify the condition checks which have data dependency on the data storing instructions. Moreover, the condition checks in Android system usually rely on dynamic values determined by run-time system status, which are hard-to-obtained by static analysis. Third, these tools (e.g., [39], [51]) are insensitive (not designed) to memory size change. It is hard for them to drive the path exploration and vulnerability exploitation with the feedback from memory size sanitizer, and achieve good performance. Last, they commonly used heavy instrumentation, for example, instruction level [34] and basic block level [50] instrumentation, which significantly affected the effectiveness of monitoring memory size change.

Other resource consumption related vulnerability. Another thread of related work leveraged DGF to find resource consumption vulnerabilities and bugs, such as algorithm complexity vulnerabilities. For example, SlowFuzz [52], PerfFuzz [53], Singularity [54], and HotFuzz [55] studied how fuzzing can be leveraged to automatically detect the time-consuming code. Specifically, SlowFuzz tracked the total count of instructions executed during a run to automatically find inputs that maximize computational resource utilization. PerfFuzz associated each program location to an input that exercised that location most and aimed to generate inputs that independently maximized the execution count of each CFG edge. Existing work aimed to find the worst inputs by monitoring the time consumed in each round of execution. However, since each straw has little influence on the memory size, even smaller than the regular change of system memory, only monitoring each round of execution is ineffective to detect straw vulnerabilities. Additionally, existing work only focused on testing a small set of programs. They used a fixed-time strategy to adjust the selection of seeds for exploration and exploitation, which is not suitable for testing straw vulnerabilities.

IX. Conclusion

In this work, we conduct the first systematic study on the straw vulnerabilities, which can cumulatively increase the memory consumption of system server by exploiting exposed data storing and cause temporary/permanent DoS attacks. We propose a directed grey-box fuzzing based approach (named `StrawFuzzer`) against the problems and discover 35 straw vulnerabilities as well as 474 vulnerable interfaces affecting 35% of real-world Android system services. Our findings show that more complete checks for data from apps are strongly needed.

X. Acknowledgment

We would like to thank our shepherd Kevin Borgolte and anonymous reviewers for their insightful comments that helped improve the quality of the paper, as well as Guangliang Yang for his help on the paper writing. This work was supported in part by National Natural Science Foundation of China (U1836210, U1836213, U1736208, 61972099, 62172105, 62102093), and Natural Science Foundation of Shanghai (19ZR1404800). Peng Liu was partially supported by NSF CNS-1814679. Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] Strawfuzzer open source address. Https://github.com/kekeLian/StrawFuzzer.
[2] Transactiontoolargeexception. Http://dwz.date/eMNP.
[3] Anr. Https://developer.android.com/topic/performance/vitals/anr.
[4] Z. Wang, B. Liblit, and T. Reps, "Tofu: Target-orienter fuzzer," arXiv preprint arXiv:2004.14375, 2020.
[5] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in 2017 IEEE Symposium on Security and Privacy (SP).
[6] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.
[7] M. Eddington, "Peach fuzzing platform," Peach Fuzzer, vol. 34, 2011.
[8] D. Vyukov, "Syzkaller," 2015.
[9] P. Wang and X. Zhou, "Sok: The progress, challenges, and perspectives of directed greybox fuzzing," arXiv preprint arXiv:2005.11907, 2020.
[10] American fuzzy lop (afl) fuzzer. Https://lcamtuf.coredump.cx/afl/.
[11] Smali. Https://github.com/JesusFreke/smali.
[12] vdex extractor. Https://github.com/anestisb/vdexExtractor.
[13] Soot. Https://github.com/Sable/soot.
[14] rovo89. Xposed framework. Http://dwz.date/eMNT.
[15] MITRE. Cwe-248: Uncaught exception. Http://dwz.date/eMNU.
[16] K. Wang, Y. Zhang, and P. Liu, "Call me back! attacks on system server and system apps in android through synchronous callback," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.
[17] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.
[18] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.
[19] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.
[20] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.
[21] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.
[22] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it:{UI} state inference and novel android attacks," in 23rd {USENIX} Security Symposium ({USENIX} Security 14).
[23] Android and rtos together: The dynamic duo for today's medical devices. Http://dwz.date/eMNF.
[24] Northrop to demo darpa navigation system on android. Https://article.wn.com/view/WNATf.
[25] Nexus one launched into space on cubesat, becomes first phonesat in orbit. Http://dwz.date/eMND.
[26] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu, "Towards discovering and understanding unexpected hazards in tailoring antivirus software for android," in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, 2015.
[27] D. Lundberg, B. Farinholt, E. Sullivan, R. Mast, S. Checkoway, S. Savage, A. C. Snoeren, and K. Levchenko, "On the security of mobile cockpit information systems," in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.
[28] Cwe-400: Uncontrolled resource consumption. Http://dwz.date/eMNW.
[29] D. Wang and J. Hoffmann, "Type-guided worst-case input generation," Proceedings of the ACM on Programming Languages, 2019.
[30] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao, "End-to-end verification of stack-space bounds for c programs," ACM SIGPLAN Notices, 2014.
[31] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).
[32] J. Vilk and E. D. Berger, "Bleak: automatically debugging memory leaks in web applications," ACM SIGPLAN Notices, 2018.
[33] M. Jump and K. S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages," in Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2007.
[34] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in 42nd International Conference on Software Engineering. ACM, 2020.
[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in USENIX Annual Technical Conference (ATC), 2012.
[36] A. Samsonov and K. Serebryany, "New features in addresssanitizer," 2013.
[37] A. Armando, A. Merlo, M. Migliardi, and L. Verderame, "Would you mind forking this process? a denial of service attack on android (and some countermeasures)," in IFIP International Information Security Conference, 2012.
[38] Y. Gu, K. Sun, P. Su, Q. Li, Y. Lu, L. Ying, and D. Feng, "Jgre: An analysis of jni global reference exhaustion vulnerabilities in android," in 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).
[39] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang, "Exception beyond exception: Crashing android system by trapping in" uncaught exception"," in 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).
[40] Fork bomb. Https://en.wikipedia.org/wiki/Forkbomb.
[41] W. Eddy et al., "Tcp syn flooding attacks and common mitigations," RFC 4987, August, Tech. Rep., 2007.
[42] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," arXiv preprint arXiv:2002.10751, 2020.
[43] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in 42nd International Conference on Software Engineering. ACM, 2020.
[44] H. Liang, Y. Zhang, Y. Yu, Z. Xie, and L. Jiang, "Sequence coverage directed greybox fuzzing," in 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC).
[45] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.
[46] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in 29th {USENIX} Security Symposium ({USENIX} Security 20).
[47] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in 2020 IEEE Symposium on Security and Privacy (SP).
[48] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in NDSS, 2019.
[49] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in 2018 IEEE Symposium on Security and Privacy (SP).

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in NDSS, 2017.
[51] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "{FANS}: Fuzzing android native system services via automated interface analysis," in 29th {USENIX} Security Symposium ({USENIX} Security 20).
[52] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.
[53] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018.
[54] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.
[55] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," arXiv preprint arXiv:2002.03416, 2020.

APPENDIX

Algorithm 1 StrawFuzzer

Input: m - Public interface
Input: t - Targeted data storing instruction
Input: i - Initial seeds with energy Initial_Energy
Output: s - test cases used for straw attacks

$s \leftarrow \emptyset$
$Explore \leftarrow i$
$Exploit \leftarrow \emptyset$
while time and resource budget do not expire **do**
  $(seed, energy) \leftarrow selectMax(Explore \cup Exploit)$
  for j from 1 to $energy$ **do**
    $seed' \leftarrow mutate(seed)$
    if $seed \in Explore$ **then**
      $distance \leftarrow explore(m, t, seed')$
      $energy = evaluate(distance)$
      if $distance == 0$ **then**
        $Exploit.add(seed', energy)$
      else
        $Explore.add(seed', energy)$
      end if
    else if $seed \in Exploit$ **then**
      $memorysize \leftarrow exploit(m, t, seed')$
      if cumulative consequence triggered **then**
        $s.add(seed')$
      else
        $energy = evaluate(memorysize)$
        $Exploit.add(seed', energy)$
      end if
    end if
  end for
  $newEnergy = anneal(seed)$
  $energyUpdate(seed, newEnergy)$
end while
return s

Algorithm 2 Call Graph Probability Estimate

Input: Entry method e and target method s
Input: Call graph $CG =< V, E >$ with e as root node
Output: $\{P_r(v)|v \in V\}$

$M = \{v|v$ could reach s on $CG\}$
for $m_i \in \{V - M\}$ **do**
  $P_r(v) = 0.0$
end for
$G' < M, E' >$= sub-graph of CG by removing $v \in \{V - M\}$
$G < M, E'' >$= DAG of G' by removing loops
vl = topological sorted G
$P_r(s) = 1.0$
for $v \leftarrow pop\_last(vl)$ **do**
  $\{P_r\{v \rightarrow v'\}|v' \in Next(v)\} = EstimateOnCFG(v)$
  $P_r(v) = \sum_{v' \in Next(v)} P_r\{v \rightarrow v'\} \cdot P_r(v')$
end for
return $\{P_r(i)|v \in M\}$

Algorithm 3 EstimateOnCFG(): Calling Probability Estimate on CFG

Input: Method e
Output: $\{P_r\{e \rightarrow v\}|v \in Next(e)\}$

$G < V = b_0, b_1, \ldots, b_n, E >$= CFG of e
$G' < V', E' >$= DAG of G by removing loops
bl = topological sorted G {Each node b has a property p represents its estimate probability}
for all $b \in V'$ **do**
  $b.p = 0.0$
end for
for all $v \in Next(e)$ **do**
  $P_r\{e \rightarrow v\} = 0.0$
end for
$b_0.p = 1.0$
for $b \leftarrow pop\_first(bl)$ **do**
  for $b' \in Next(b)$ **do**
    $b'.p = b'.p + b.p/length(Next(b))$
  end for
  for all v called in b **do**
    $P_r\{e \rightarrow v\} = max(P_r\{e \rightarrow v\}, b.p)$
  end for
end for
for all $v \in M$ **do**
  $P_r\{e \rightarrow v\} = P_r\{e \rightarrow v\} / \sum_{v' \in M} P_r\{e \rightarrow v'\}$
end for

Table III: The exploitable vulnerabilities exposed by `StrawFuzzer`. The second column lists the methods containing unique exploitable data storing instructions in the Android framework. In the fourth column, ① stands for the Pixel 3 (Android10.0), ② stands for the Pixel 3XL (Android11.0) and ③ stands for the OnePlus 7 (Android10.0). The last column lists the number of public interfaces provided by system services which can be used to trigger the vulnerability by an Android app. The interfaces tagged with ⋆ can be used to launch permanent DoS attacks. In particular, methods in the deserialization phase (i.e., methods in *android.os.Parcel* class) are aggregated into two lines according to the functional similarity.

ID	Method Containing Unique Exploitable Data Storing Instructions	Detail	Affected Systems	#Affected Interfaces
Unlimited Data Storing:				
1	Account.Account()	insert user's online accounts into *accounts_de.db* and cache them in account service. Apps that register the accounts can access online resources without entering credentials repeatedly.	①②③	276
2	AccountManagerService.insertDeAccount()	insert user's online accounts into *accounts_de.db* and cache them in account service. Apps that register the accounts can access online resources without entering credentials repeatedly.	①②③	2⋆
3	Settings$NameValueCache.getStringForUser()	cache the name and result of query to the system settings databse in system server, which can improve the efficiency of subsequent queries.	①②③	2
4	PlaybackActivityMonitor.trackPlayer()	store the configuration of audio players in audio service, which can be used to track them and their states.	①②③	1
5	WifiMulticastLockManager.acquireLock()	store the acquired wifi multicast lock and its caller name in wifi service. The lock is used to allow an app to receive wifi multicast packets.	①②③	1
6	UsageStatsService$BinderService.reportPastUsageStart()	store the Activity and associated token string in usagestats service for usage statistics recording. The token string is defined by app to represent the usage of in-app features.	①②③	2
7	AppTimeLimitController.noteUsageStart()	record the Usage entity name in usagestats service when an entity become active. The Usage entity is responsible for aggregating application usage data.	①②③	2
8	BroadcastDispatcher.enqueueOrderedBroadcastLocked()	add broadcasts into an orderly scheduling queue maintained by activity service.	①②③	2
9	GnssManagerService.addGnssDataListenerLocked()	store the registeded GNSS data listener and information of the caller in location service.	②	3
10	ActivityManagerService.broadcastIntentLocked()	keep the broadcast data in activity service if the broadcast is labeled as sticky. The sticky broadcast will be maintained in the system for a period of time.	①②③	2
11	RemoteCallbackList.register()	store the registered remote callback and associated information in system server.	①③	1
12	PlaybackActivityMonitor$PlayMonitorClient.init()	register a callback in audio service to track the playback activity listener.	①②③	1
13	RecordingActivityMonitor$RecorderDeathHandler.init()	register a callback in audio service to track the audio recorder provided by the client.	①②③	1
14	RecordingActivityMonitor$RecMonitorClient.init()	register a callback in audio service to track the listener which is used to monitor the audio recording updates.	①②③	1
15	Session.Session()	register a callback in window service to track the initialized window session. This session is used to interact with window manager.	①②③	1
16	SensorPrivacyService$DeathRecipient.DeathRecipient()	register a callback in sensor_privacy service to track the registered sensor privacy state listener.	①②③	1
17	WifiP2pServiceImpl.getMessenger()	register a callback in wifip2p service to track the Messenger, which is used by client apps to establish asynchronous communication with wifip2p service.	①②③	1
18	AudioPlaybackConfiguration$IPlayerShell.monitorDeath()	register a callback in audio service to monitor the corresponding audio player.	①②③	1
19	LocationManagerService.linkToListenerDeathNotificationLocked()	register a callback in location service for the provided listener. The listener is registered by client apps to monitor the dnss data changes.	①②③	3
20	SecrecyService$zta.zta()	register a callback in secrecy service for the provided secrecy service receiver. The receiver is registered by client apps to track the secrecy service state changes.	③	1
21-24	Parcel.readStringList() Parcel.readTypedList() Parcel.readParcelableList() Parcel.readListInternal()	deserialize data from client inputs transmitted via IPC and add them into a list continuously.	①②③	66
25-35	Parcel.createStringArrayList() Parcel.createTypedArrayList() Parcel.readStringArray() Parcel.readSparseArray() Parcel.readCharSequenceArray() Parcel.readSparseBooleanArray() Parcel.readParcelableArray() Parcel.readArray() Parcel.createStringArray() Parcel.readArraySet() Parcel.readHashMap()	create an array whose size is read from client inputs transmitted via IPC.	①②③	245
Uncaught Exception:				
1	RemoteCallbackList.register()	invoke method 'IInterface.asBinder()' on a null object reference	①②③	1
2	ActivityManagerService.handleApplicationWtfInner()	read field 'exceptionMessage' on a null object reference	①②③	2
3	StringBuilder.StringBuilder()	invoke method 'String.length()' on a null object reference	①②③	1
4	IBluetoothMidiService.addBluetoothDevice()	invoke method 'MidiDeviceService.asBinder()' on a null object reference	①②③	1
5	UserController.ensureNotSpecialUser()	throw IllegalArgumentException if user id less than 0	③	1

Table IV: Exploitable interfaces discovered by `StrawFuzzer`.

ID	Service	Exploitable Interfaces	#
		Unlimited Data Storing	
0	accessibility	registerUiTestAutomationService, sendAccessibilityEvent	2
1	account	startAddAccountSession, getAccountVisibility, getPassword, clearPassword, renameAccount, getPreviousName, isCredentialsUpdateSuggested, hasFeatures, addAccountExplicitlyWithVisibility, updateCredentials, removeAccountAsUser, getPackagesAndVisibilityForAccount, getAccountByTypeAndFeatures, updateAppPermission, unregisterAccountListener, addAccountAsUser, renameSharedAccountAsUser, copyAccountToUser, getAccountsByFeatures, setAuthToken, hasAccountAccess, someUserHasAccount, getAuthToken, startUpdateCredentialsSession, getUserData, registerAccountListener, confirmCredentialsAsUser, peekAuthToken, createRequestAccountAccessIntentSenderAsUser, setAccountVisibility, addAccount, removeAccountExplicitly, removeSharedAccountAsUser, accountAuthenticated, removeAccount, addAccountExplicitly, setUserData, setPassword	38
2	activity	handleApplicationStrictModeViolation, startActivityWithFeature, publishContentProviders, updateLockTaskPackages, startDelegateShellPermissionIdentity, broadcastIntent, bindService, bindIsolatedService, publishService, registerReceiver, noteAlarmStart, scheduleApplicationInfoChanged, getIntentSenderWithFeature, noteAlarmFinish, startActivity, registerReceiverWithFeature, startService, startConfirmDeviceCredentialIntent, noteWakeupAlarm, updatePersistentConfiguration, peekService, updateConfiguration, broadcastIntentWithFeature, unbindBackupAgent, startActivityAsUserWithFeature, setServiceForeground, startActivityAsUser, startRecentsActivity, stopService, finishActivity, sendIntentSender, unbindFinished, unbroadcastIntent, getIntentSender	34
3	activity_task	reportAssistContextExtras, navigateUpTo, clearLaunchParamsForPackages, startVoiceActivity, enterPictureInPictureMode, startActivityAndWait, startDreamActivity, startActivities, addAppTask, startActivityAsUser, startActivityWithConfig, startActivity, updateConfiguration, updateDisplayOverrideConfiguration, activityIdle, startActivityAsCaller, setPictureInPictureParams, startAssistantActivity, launchAssistIntent, startActivityIntentSender, startNextMatchingActivity, finishActivity, startRecentsActivity, isActivityStartAllowedOnDisplay	24
4	alarm	set	1
5	android.security.keystore	generateKey, begin, update, importWrappedKey, importKey, attestDeviceIds, attestKey, finish	8
6	app_prediction	notifyAppTargetEvent	1
7	appops	getHistoricalOps, setAudioRestriction, getHistoricalOpsFromDiskRaw, addHistoricalOps	4
8	appwidget	bindRemoteViewsService, partiallyUpdateAppWidgetIds, updateAppWidgetIds, updateAppWidgetProvider	4
9	audio	getFocusRampTimeMs, getMinVolumeIndexForAttributes, dispatchFocusChange, getDevicesForAttributes, getVolumeIndexForAttributes, getMaxVolumeIndexForAttributes, abandonAudioFocus, setUidDeviceAffinity, setFocusRequestResultFromExtPolicy, setUserIdDeviceAffinity, playerAttributes, trackPlayer, setVolumeIndexForAttributes, trackRecorder, registerRecordingCallback, registerPlaybackCallback, startWatchingRoutes, requestAudioFocus	18
10	autofill	setAutofillFailure, setUserData, setAugmentedAutofillWhitelist	3
11	backup	initializeTransportsForUser, updateTransportAttributesForUser, excludeKeysFromRestore, requestBackup, adbBackup, filterAppsEligibleForBackupForUser, requestBackupForUser, fullTransportBackupForUser	8
12	batterystats	noteWifiScanStoppedFromSource, notePhoneSignalStrength, noteBleScanStarted, noteLongPartialWakelockFinishFromSource, noteWifiRunning, noteStartWakelockFromSource, noteWifiStopped, noteFullWifiLockAcquiredFromSource, noteWifiBatchedScanStartedFromSource, noteStopWakelockFromSource, noteLongPartialWakelockStartFromSource, noteWifiRunningChanged, noteChangeWakelockFromSource, noteBleScanResults, noteFullWifiLockReleasedFromSource, noteGpsChanged, noteWifiBatchedScanStoppedFromSource, noteBleScanStopped, noteWifiScanStartedFromSource	19
13	bluetooth_manager	registerStateChangeCallback, registerAdapter	2
14	clipboard	setPrimaryClip, addPrimaryClipChangedListener	2
15	companiondevice	associate, stopScan	2
16	connectivity	pendingListenForNetwork, establishVpn, provisionVpnProfile, setAlwaysOnVpnPackage, releaseNetworkRequest, startLegacyVpn, requestNetwork, listenForNetwork, setGlobalProxy, declareNetworkRequestUnfulfillable, registerNetworkAgent, registerConnectivityDiagnosticsCallback, pendingRequestForNetwork, startTethering	14
17	connmetrics	logEvent	1
18	content	cancelSync, getPeriodicSyncs, getIsSyncableAsUser, requestSync, isSyncPending, cancelRequest, setSyncAutomaticallyAsUser, isSyncPendingAsUser, syncAsUser, addPeriodicSync, setSyncAutomatically, cancelSyncAsUser, addStatusChangeListener, getSyncStatusAsUser, getSyncAutomatically, getSyncStatus, sync, setIsSyncableAsUser, removePeriodicSync, setIsSyncable, getSyncAutomaticallyAsUser, isSyncActive, getIsSyncable	23
19	content_capture	removeData	1
20	content_suggestions	classifyContentSelections	1
21	country_detector	addCountryListener	1
22	crossprofileapps	resetInteractAcrossProfilesAppOps, startActivityAsUserByIntent	2
23	device_policy	enableSystemAppWithIntent, setUserRestriction, setLockTaskPackages, setDelegatedScopes, addPersistentPreferredActivity, setKeepUninstalledPackages, setCrossProfilePackages, setAlwaysOnVpnPackage, setPermittedAccessibilityServices, setAffiliationIds, setRecommendedGlobalProxy, setPackagesSuspended, updateOverrideApn, addOverrideApn, setPermittedCrossProfileNotificationListeners, setMeteredDataDisabledPackages, setUserControlDisabledPackages, setPermittedInputMethods, uninstallCaCerts, addCrossProfileIntentFilter, startManagedQuickContact, generateKeyPair, setCrossProfileCalendarPackages	23
24	deviceidle	addPowerSaveWhitelistApps, registerMaintenanceActivityListener	2
25	dropbox	add, isTagEnabled	2
26	econtroller	continueOperation, getDownloadableSubscriptionMetadata, downloadSubscription, setSupportedCountries	4
27	ethernet	setConfiguration, addListener	2
28	extphone	setLteBandPriority	1
29	fingerprint	authenticate, addLockoutResetCallback	2
30	imms	updateStoredMessageStatus	1
31	input_method	startInputOrWindowGainedFocus, addClient	2
32	ions	updateAvailableNetworks	1
33	ipsec	createTransform	1
34	ircs	getRcsThreads, storeFileTransfer	2
35	isms	sendStoredMultipartText, sendMultipartTextForSubscriberWithOptions, sendMultipartTextForSubscriber	3
36	jobscheduler	scheduleAsPackage, schedule, enqueue	3
37	launcherapps	cacheShortcuts, startSessionDetailsActivityAsUser, registerShortcutChangeCallback, pinShortcuts, uncacheShortcuts, getShortcuts	6
38	location	addGnssNavigationMessageListener, addGnssAntennaInfoListener, addGnssMeasurementsListener, injectGnssMeasurementCorrections, registerGnssStatusCallback	5
39	media_router	deselectRouteWithRouter2, setDiscoveryRequestWithRouter2, selectRouteWithRouter2, requestCreateSessionWithManager, transferToRouteWithManager, setRouteVolumeWithRouter2, setRouteVolumeWithManager, selectRouteWithManager, deselectRouteWithManager, requestCreateSessionWithRouter2, transferToRouteWithRouter2	11

ID	Service	Exploitable Interfaces	#
40	media_session	notifySession2Created	1
41	midi	openDevice, setDeviceStatus, getDeviceStatus, registerDeviceServer	4
42	mount	registerListener	1
43	netpolicy	snoozeLimit, getNetworkQuotaInfo	2
44	netstats	registerUsageCallback, getDetailedUidStats, unregisterUsageRequest	3
45	network_management	addInterfaceToLocalNetwork, removeRoute, registerNetworkActivityListener, startTetheringWithConfiguration, removeRoutesFromLocalNetwork, setInterfaceConfig, setDnsForwarders, getNetworkStatsUidDetail, addLegacyRouteForNetId, addRoute	10
46	nfc	setForegroundDispatch, invokeBeamInternal	2
47	notification	enqueueNotificationWithTag, cancelNotificationsFromListener, setAutomaticZenRuleState, setNotificationsShownFromListener, updateNotificationChannelForPackage, addAutomaticZenRule, applyAdjustmentsFromAssistant, createConversationNotificationChannelForPackage, updateAutomaticZenRule, updateNotificationChannelGroupForPackage, getActiveNotificationsFromListener, updateNotificationChannelFromPrivilegedListener, updateNotificationChannelGroupFromPrivilegedListener	13
48	oneplus_nfc_service	setSupportCardTypes	1
49	opscenecallblock	isNotificationMutedByESport, isMutedByCallBlocker, isCallBlockedWithUid	3
50	package	grantDefaultPermissionsToEnabledImsServices, queryIntentActivityOptions, querySyncProviders, queryIntentContentProviders, activitySupportsIntent, getLastChosenActivity, setDistractingPackageRestrictionsAsUser, resolveService, setMimeGroup, notifyDexLoad, revokeDefaultPermissionsFromDisabledTelephonyDataServices, addPersistentPreferredActivity, canForwardTo, getPackagesHoldingPermissions, queryIntentReceivers, notifyPackagesReplacedReceived, grantDefaultPermissionsToEnabledTelephonyDataServices, queryIntentActivities, installExistingPackageAsUser, revokeDefaultPermissionsFromLuiApps, addCrossProfileIntentFilter, grantDefaultPermissionsToEnabledCarrierApps, replacePreferredActivity, runBackgroundDexoptJob, resolveIntent, currentToCanonicalPackageNames, verifyIntentFilter, findPersistentPreferredActivity, getUnsuspendablePackagesForUser, setPackagesSuspendedAsUser, addPreferredActivity, canonicalToCurrentPackageNames, queryIntentServices, setLastChosenActivity	34
51	package_native	isAudioPlaybackCaptureAllowed	1
52	permissionmgr	grantDefaultPermissionsToEnabledImsServices, grantDefaultPermissionsToEnabledCarrierApps, revokeDefaultPermissionsFromLuiApps, revokeDefaultPermissionsFromDisabledTelephonyDataServices, grantDefaultPermissionsToEnabledTelephonyDataServices	5
53	phone	setSystemSelectionChannels, setAllowedCarriers, enableVisualVoicemailSmsFilter, requestCellInfoUpdateWithWorkSource, setRoamingOverride, getSubIdForPhoneAccount, updateEmergencyNumberListTestMode, requestNetworkScan, getCarrierPackageNamesForIntentAndPhone	9
54	platform_compat	isChangeEnabled, getAppConfig, reportChange	3
55	power	acquireWakeLock, updateWakeLockWorkSource	2
56	print	startPrinterDiscovery, validatePrinters, startPrinterStateTracking, stopPrinterStateTracking, getCustomPrinterIcon	5
57	role	setRoleNamesFromController	1
58	secrecy	isInEncryptedAppList, registerSecrecyServiceReceiver	2
59	sensor_privacy	addSensorPrivacyListener	1
60	shortcut	removeLongLivedShortcuts, disableShortcuts, getShareTargets, enableShortcuts, removeDynamicShortcuts, pushDynamicShortcut	6
61	simphonebook	updateAdnRecordsWithContentValuesInEfBySearchUsingSubId	1
62	slice	checkSlicePermission	1
63	telecom	handleCallIntent, startConference, registerPhoneAccount	3
64	telephony.registry	notifyBarringInfoChanged, notifyDataConnectionForSubscriber, notifyCellLocation, notifyDataConnection, notifyPhysicalChannelConfigurationForSubscriber, notifyServiceStateForPhoneId, notifySignalStrengthForPhoneId, notifyOutgoingEmergencySms, notifyPhysicalChannelConfiguration, notifyCellInfo, notifyPhoneCapabilityChanged, notifyOutgoingEmergencyCall, notifyCellLocationForSubscriber, notifyCellInfoForSubscriber, notifyRegistrationFailed	15
65	telephony_ims	requestCapabilities	1
66	textclassification	onSelectionEvent, onDetectLanguage, onCreateTextClassificationSession, onSuggestConversationActions, onTextClassifierEvent, onClassifyText, onSuggestSelection, onGenerateLinks	8
67	thermalservice	registerThermalStatusListener	1
68	usagestats	registerAppUsageObserver, registerAppUsageLimitObserver, registerUsageSessionObserver, reportUsageStart, reportPastUsageStart, reportChooserSelection	6
69	usb	addDevicePackagesToPreferenceDenied, setDevicePersistentPermission, setDevicePackage, addAccessoryPackagesToPreferenceDenied, removeDevicePackagesFromPreferenceDenied, removeAccessoryPackagesFromPreferenceDenied	6
70	user	createProfileForUser, createProfileForUserEvenWhenDisallowedWithThrow, createProfileForUserEvenWhenDisallowed, createProfileForUserWithThrow	4
71	vibrator	vibrate, setAlwaysOnEffect	2
72	voiceinteraction	getActiveServiceSupportedActions, startVoiceActivity, finish, startAssistantActivity	4
73	wifi	getMatchingScanResults, connect, getMatchingPasspointConfigsForOsuProviders, getAllMatchingPasspointProfilesForScanResults, addNetworkSuggestions, startSubscriptionProvisioning, removeNetworkSuggestions, getMatchingOsuProviders, getAllMatchingFqdnsForScanResults, setWifiApConfiguration, addOrUpdatePasspointConfiguration, startLocalOnlyHotspot, startTetheredHotspot, setSoftApConfiguration, getWifiConfigsForPasspointProfiles, acquireMulticastLock, updateWifiLockWorkSource, acquireWifiLock, getWifiConfigForMatchedNetworkSuggestionsSharedWithUser, startSoftAp, addOrUpdateNetwork, save	22
74	wifiaware	requestMacAddresses	1
75	wifip2p	getMessenger	1
76	wifirtt	cancelRanging, startRanging	2
77	window	modifyDisplayWindowInsets, openSession, watchRotation	3
		Uncaught Exception	
1	activity	handleApplicationWtf	1
2	bluetooth_manager	bindBluetoothProfileService	1
3	content	sync	1
4	fingerprint	addLockoutResetCallback	1
5	media_session	dispatchAdjustVolume	1
6	oneplus_colordisplay_service	notifySetUp	1