Ghost in the Binder: Binder Transaction Redirection Attacks in Android System Services

Xiaobo Xiang
xiangxiaobo@iie.ac.cn
CAS-KLONAT[§], IIE, CAS[†]
School of Cyber Security, UCAS[¶]
Alpha Lab[*], 360 GESG[‡]
Beijing, China

Ren Zhang
ren@nervos.org
Nervos
Shandong Institute of Blockchain
Beijing, China

Hanxiang Wen
arnow117@gmail.com
Ant Group
Hangzhou, China

Xiaorui Gong [✉]
gongxiaorui@iie.ac.cn
CAS-KLONAT[§], IIE, CAS[†]
School of Cyber Security, UCAS[¶]
Beijing, China

Baoxu Liu
liubaoxu@iie.ac.cn
CAS-KLONAT[§], IIE, CAS[†]
School of Cyber Security, UCAS[¶]
Beijing, China

ABSTRACT

Binder, the main mechanism for Android applications to access system services, adopts a client-server role model in its design, assuming the system service as the server and the application as the client. However, a growing number of scenarios require the system service to act as a Binder client and to send queries to a Binder server possibly instantiated by the application. Departing from this role-reversal possibility, this paper proposes the Binder Transaction Redirection (BiTRe) attacks, where the attacker induces the system service to transact with a customized Binder server and then attacks from the Binder server—an often unprotected direction. We demonstrate the scale of the attack surface by enumerating the utilizable Binder interfaces in BiTRe, and discover that the attack surface grows with the Android release version. In Android 11, more than 70% of the Binder interfaces are affected by or can be utilized in BiTRe. We prove the attacks' feasibility by (1) constructing a prototype system that can automatically generate executable programs to reach a substantial part of the attack surface, and (2) identifying a series of vulnerabilities, which are acknowledged by Google and assigned ten CVEs.

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**.

[✉] Corresponding author.
[§] Key Laboratory of Network Assessment Technology, CAS.
[†] Institute of Information Engineering, Chinese Academy of Sciences.
[¶] University of Chinese Academy of Sciences.
[‡] 360 Government and Enterprise Security Group.
[*] Part of this work was done during Xiaobo Xiang's research internship at Alpha Lab.

KEYWORDS

mobile security; Android security; vulnerability analysis;

ACM Reference Format:
Xiaobo Xiang, Ren Zhang, Hanxiang Wen, Xiaorui Gong [✉], and Baoxu Liu. 2021. Ghost in the Binder: Binder Transaction Redirection Attacks in Android System Services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3485390

1 INTRODUCTION

Residing at the core of Android systems, *Android system services* often execute in privileged processes and have access to the users' private information and critical system resources, thus constituting highly attractive targets for attackers. Most system services can only be accessed via *Binder*, an Inter-Process Communication (IPC) mechanism introduced since the original release of Android. In a typical Binder transaction, the application, acting as a *Binder client*, sends requests to a system service, acting as a *Binder server*; the latter then checks the clients' identities and permissions, processes the requests, and responds to the client.

Given the crucial role of Binder, its security has been constantly scrutinized by both academia and industry. Most attacks against Android system services via Binder can be categorized into three types based on the exploited vulnerabilities. The first type discovers [2, 12, 13, 19] and utilizes [1, 18, 22, 23, 39] misconfigured *permission checks* to visit critical system resources without the access rights. The second type targets Binder's *data serialization and deserialization processes* [15, 38]. For example, the XBRF attack proposed by Rosa [38] allows the attacker to indirectly invoke some protected objects by sending malformed data to the system service, who mistakenly deserializes these data as references to these protected objects and initiates the invocation. The third type [16, 21, 32] exploits the vulnerabilities in the system services' *input validation*, by also sending malformed data to trigger logical or memory corruption bugs. Although different in mechanisms, all three types share a common pattern in their strategies: the attacker application is a Binder client, and the victim system service is a Binder server.

Noticeably, recent Android releases more frequently require system services to *act as Binder clients* to provide certain functionalities—e.g., to send intermediate results to the application, or to request services from another system service or application. In this case, an application transacts a *Binder proxy* referencing a Binder server to the system service, through which the system service can interact with the Binder server. It is generally believed that this mechanism also strengthens the security of Android [29], as a cumbersome system service can now be split into multiple ones that operate independently of each other, implementing the "separation of duty" principle.

In this study, we, alternatively, indicate that there are significant security flaws embedded in this mechanism. Specifically, when the system services function as Binder clients, the applications, including malicious ones, are allowed to assume the Binder-server role, attacking the system service from an often unprotected direction. We thus forward a new and common family of attacks exploiting this mechanism, called the *Binder Transaction Redirection (BiTRe)* attacks, based on the following two observations: (1) in the *role-reversal case* enabled by the current design, the Binder proxy, through which the system service interacts with the Binder server, can be implemented by an application—therefore an attacker; (2) there is no universal way to authenticate the identity of a Binder server. These design choices allow an attacker to customize an evil Binder server and then to send its proxy to the system service, thus redirecting the system service to interact with the customized Binder server. Lacking explicit acknowledgment of its security implications, attacks leveraging the role-reversal case [24, 42] remain largely unknown and unaddressed. We demonstrate that the attack surface is not only broader but also more damaging than previously believed, as the vulnerabilities lead to, in the worst case, the attacker's privilege escalated to that of the user "system", with all Android permissions granted.

Specifically, there are three key steps in a BiTRe attack after constructing and instantiating the customized Binder server: (a) the attacker application—still a Binder client—interacts with the system service by invoking the latter's functions in the correct order, until it accepts the Binder proxy corresponding to the customized Binder server; (b) the application calls certain functions in the system service to trigger a transaction between the latter and the customized Binder server via the proxy; (c) the Binder server, also residing in the attacker application, replies with malformed data to attack the system service.

We systematically analyze the BiTRe attacks with the following technical efforts:

Enumerating the Potential BiTRe Targets in Android System Services. We first measure the scale of the attack surface via an LLVM compiler [30] plugin by counting two types of Binder interfaces: (1) *target Binder interface (TBI)*, which are reachable system services that can receive Binder proxies from applications; (2) *customizable Binder interface (CBI)*, which can be instantiated by the attacker and transact with TBIs as Binder servers. Our enumeration demonstrates that these numbers grow with the Android release version, posing continuous challenges to security auditors. Among the 176 Binder interfaces in Android 11—the latest version as of May 2021, 57 can serve as TBIs and 84 can serve as CBIs, resulting in a total number of 128 as 13 can serve as both.

Quantifying the Attack Capabilities of CBIs. CBIs are the attacker's tools in BiTRe. To prioritize our efforts in locating the vulnerabilities, we rank all CBIs based on an *Interface Complexity* metric we defined and measured. A CBI's Interface Complexity is calculated by (1) grouping the inputs and outputs of the CBI's member functions according to their data types, (2) assigning each data type with a weight value, and (3) computing the weighted counts of these inputs and outputs. A higher Interface Complexity indicates higher potential damage if the CBI is crafted by an attacker. This metric proves indicative: of the ten CVEs corresponding to this study, four of them involve CBIs with the top four Interface Complexity.

Confirming the Attacks' Feasibility. We divide a BiTRe attack into two phases and confirm their feasibility separately: a *preparation phase* when the attacker process transacts with a TBI server and induces it to transact with the CBI server—the aforementioned steps (a) and (b), and an *attacking phase* when the CBI server launches attacks to cause damage—step (c).

We demonstrate the efficiency of the preparation phase by building a prototype system, which generates executable Proofs of Reachability (PoRs) to automatically induce BiTRe transactions. Our PoRs do not trigger or exploit vulnerabilities, but merely redirect system services to transact with our customized Binder servers. The key challenge is to set the TBI server in two specific states: the first to accept the Binder proxy, the second to transact with the CBI. This requires the prototype system to locate the correct TBI-function-calling sequence and the syntactically-and-semantically-correct function parameters. We address this challenge by guiding the system with the knowledge we extracted from our TBI/CBI enumeration. Within 24 hours of its execution, our system generates 81 successful PoRs, which differ from each other either in the TBI or the called CBI function. These PoRs involve 28 CBIs and 12 TBIs, proving that the attack surface is reachable with little effort.

As for the attacking phase, we perform static code review and dynamic security tests on role-reversal Binder transactions, starting from CBIs with high Interface Complexity. We discover a series of vulnerabilities, which are reported to Google and then patched in subsequent releases of Android. Ten CVEs are assigned to these vulnerabilities. Our PoRs can assist in triggering vulnerabilities associated with seven CVEs.

The vulnerabilities, mainly input validation errors and logical bugs, exhibit a wide range of trigger mechanisms, as some attack portals are hidden in nested data structures. We classify them into four types, based on the numbers of TBIs and CBIs involved, and elaborate their typical processes.

Identifying the Underlying Mechanism that Enables the BiTRe Attacks. BiTRe is fundamentally different from existing attacks as the malformed messages are sent from a Binder server. It cannot be solved by strengthening the permission checks, because the system service can only check the permission set of the application, rather than that of the CBI server, which is unrelated to the former. Moreover, unlike the XBRF attack, BiTRe does not rely on any data deserialization errors. Yet the worrisome scale of the

BiTRe attacks indicates some missing pieces in Android's security policies and the security auditors' efforts.

We attribute BiTRe to the implicit *fixed-role assumption* that a Binder server is a system service. Android's built-in security model adopts this assumption and only allows Binder servers to authenticate their clients' identities and permissions; clients, on the other hand, do not have the same tools. Moreover, as pointed out by Alkhalaf et al. [4], in a typical system following the client-server architecture, input validation on the server side focuses on security, whereas that on the client side focuses only on responsiveness, leaving the client not thoroughly protected.

To sum up, our contribution includes:

(1) Theoretically, we propose the BiTRe attacks, which exploit the role-reversal mechanism in Android system services.
(2) Empirically, we enumerate the potential BiTRe attack targets and measure their attack capabilities, revealing the worrisome scale of this newly discovered attack surface.
(3) Practically, we confirm the damage of the BiTRe attacks by automatically inducing BiTRe transactions and discovering a considerable number of vulnerabilities, and assist Google in fixing them.

With this study, we also call for special attention to a larger family of attacks represented by BiTRe, where the attack surfaces are overlooked by both academia and industry as they are concealed in the general assumptions defined in the system's security model.

2 DISSECTING BINDER

Binder is the default IPC mechanism for Android applications to access most system services. Here we first overview its architecture, then introduce its data structures and transaction workflow, highlighting the *active object transmission feature* that enables the role-reversal case.

2.1 Binder Architecture

On the highest level, Binder applies the Proxy design pattern [45] over a C/S role model. This architecture simplifies both the system service invocation instruction and the security policy enforcement, achieving a good balance between security and usability.

C/S Role Model. In analogy to the classic C/S architecture where a client initiates the connection to a server, in Binder, the process initiating the transaction is called the *Binder client*, and the process receiving the transaction request is called the *Binder server*. In a typical Binder transaction, the Binder client is an application, and the Binder server is a system service. The application initiates the transaction by calling system-provided APIs, which wrap the application as the Binder client and communicate with the system service. The system service, with its native Binder-server implementation, then receives the request, performs permission checks, processes the request, and sends the response.

The security model of Binder also follows the classic C/S role model, where the server is deemed more trustworthy than the client in both authentication and input validation. System-provided authentication APIs, i.e., *checkCallingPermission* [8], are only available to the Binder server, but not to the Binder clients. This *one-way*

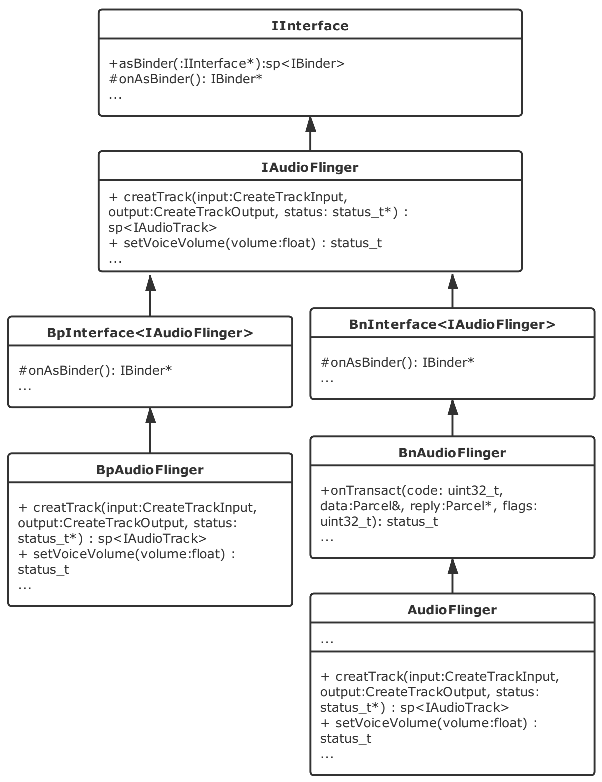
IInterface
+asBinder(:IInterface*):sp<IBinder>
#onAsBinder(): IBinder*
...
IAudioFlinger
+ creatTrack(input:CreateTrackInput, output:CreateTrackOutput, status: status_t*) : sp<IAudioTrack>
+ setVoiceVolume(volume:float) : status_t
...
BpInterface<IAudioFlinger>
#onAsBinder(): IBinder*
...
BnInterface<IAudioFlinger>
#onAsBinder(): IBinder*
...
BpAudioFlinger
+ creatTrack(input:CreateTrackInput, output:CreateTrackOutput, status: status_t*) : sp<IAudioTrack>
+ setVoiceVolume(volume:float) : status_t
...
BnAudioFlinger
+onTransact(code: uint32_t, data:Parcel&, reply:Parcel*, flags: uint32_t): status_t
...
AudioFlinger
...
+ creatTrack(input:CreateTrackInput, output:CreateTrackOutput, status: status_t*) : sp<IAudioTrack>
+ setVoiceVolume(volume:float) : status_t
...

Figure 1: Class inheritance rules in C++ Binder, with the Binder interface *IAudioFlinger* from Android-11.0.0_r1 as an example. The Binder interface inherits *IInterface*. The Binder client operates the proxy via *BpAudioFlinger*, which inherits the Binder interface through a template class. The Binder server, implemented in *BnAudioFlinger* and its derived class *AudioFlinger*, inherits the same Binder interface through another template class. Functions declared in the Binder interface are implemented on both sides.

authentication policy prevents information leakage from the system services, which often execute in privileged system processes, to applications, which execute in the *untrusted_app* domain [17]. As for input validation, we discover that Binder's client-side input validation often ignores security-related checks—a pattern also observed in other systems with the C/S architecture [4].

Proxy Design Pattern. Binder follows the *Proxy design pattern* to simplify the procedure of a client accessing a remote object, which may be in another process. By visiting the object via the proxy, the client can remain ignorant of where the object resides and how it operates. To establish the proxy-object mapping, the proxy and the object must implement the same interface.

We highlight some class naming and inheritance rules in C++ Binder that are relevant to this study. All classes implementing Binder proxies share a prefix *Bp* in their names, which stands for "Binder proxy"; all classes implementing the object, i.e., the Binder server, share a prefix *Bn*, which stands for "Binder native". A *Bp* class and its corresponding *Bn* class must implement the same

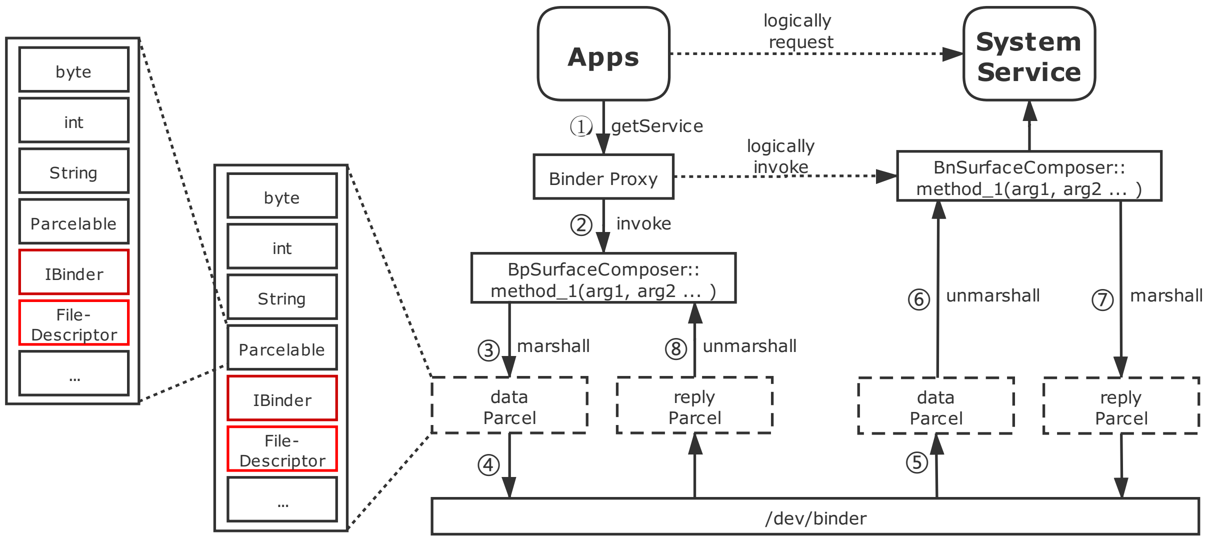

Figure 2: The Binder transaction workflow and the Parcel object.

Binder interface. In other words, a Binder client can call functions declared in its Binder interface to get served, whose actual implementation is in the Binder server. All Binder interfaces must inherit the class *IInterface*. These rules are illustrated via an example in Fig. 1. Note that C++ Binder's naming rules are different from those of Java's, which uses "Stub" and "Proxy" as the server's and the client's identifiers, respectively.

2.2 Data Structures in Binder

We overview the construction and components of the *Parcel object*, which is used to transfer data between the Binder client and its server. Parcel is a flexible container class that can encapsulate several categories of data, including (1) primitive types—such as integer and byte, (2) active objects, and (3) Parcelables. The latter two will be explained next.

Active Objects. An *active object* is not written in a Parcel object as its raw content, but rather as a special token referencing it [9]. Therefore, upon receiving the active object, a remote process can operate directly on the original object. Active objects are used to transfer a file handler as a *FileDescriptor* object, or a Binder server or a Binder proxy/client as an *IBinder* object.

Instances of a Binder interface can be cast to *IBinder* objects through calling the *asBinder* or *onAsBinder* functions declared in *IInterface*. For example, in Fig. 1, instances of *BpAudioFlinger* and *AudioFlinger* can be cast as they are derived from *IAudioFlinger*, a Binder interface.

We describe a concrete example in a crucial system service, i.e., the Service Manager service, to illustrate that both the Binder proxy and the Binder server can be cast to *IBinder* objects and transmitted via the IPC. Implementing the *IServiceManager* Binder interface, *ServiceManager* is a system service in charge of the registration and lookup of all other system services. The Binder proxy of *ServiceManager* is available to all processes. When the system is launched, these system services call the *addService* function declared in *IServiceManager* and send their Binder server instances—wrapped as *IBinder* objects. The Service Manager then adds these *IBinder* objects to its global map. Similarly, to access a system service, an application queries the name of the system service via function *getService* or *checkService* declared in *IServiceManager*. The Service Manager then returns the Binder proxy as an *IBinder* object referencing the registered Binder server.

Parcelables. A Parcel object can encapsulate another potentially complicated class instance that supports the *Parcelable protocol* [26]. To support the Parcelable protocol, a class should implement the *Parcelable interface* and two of its member functions, *readFromParcel* and *writeToParcel*, whose purposes are obvious from their names. It is worth noting that the Parcel class itself supports the Parcelable protocol. The potentially nested structure of the Parcel objects brings additional challenges to our attack surface enumeration.

2.3 Binder Transaction Workflow

We start with the transaction workflow in typical client/server roles—the application as the Binder client and the system service as the Binder server. The role-reversal case, explained afterward, only differs marginally.

Typical Workflow. As shown in Fig. 2, to access a system service, an application invokes the *getService* method in *ServiceManager*, which returns a proxy referencing the Binder server (①). When the application calls a function in the Binder server via the proxy (②), it first encapsulates the parameters into a Parcel object (③). The procedure of serializing a list of data items into a Parcel object is called *marshalling*. Likewise, the deserializing process is called *unmarshalling*. After the parameters are marshalled, the proxy sends (④), among other information, the Parcel object to the *kernel*, which delivers the request to the destination process. Upon receiving the request (⑤), the Binder server unmarshalls the parameters (⑥), delegates the request to the responsible function, marshalls the reply (⑦), and sends it back to the proxy through the kernel for unmarshalling (⑧).

Other Binder-Proxy Construction Methods. The role-reversal case differs from the default case only in ①. In this case, the application sends a Binder proxy to the system service, requesting the latter to transact with the corresponding Binder server via the proxy. The proxy can be constructed either by the application or via a system-provided API.

A system service can also construct a Binder proxy, bypassing *ServiceManager*. In this case, the system service generates a second Binder server, called a *sub-interface*, which enjoys the same privilege as the parent service. The parent service then sends the Binder proxy of the sub-interface to the application, enabling them to communicate directly.

Pervasiveness of the Role-Reversal Case. The role-reversal case provides specific features that are widely utilized in both system services and applications. Here are two common cases where a system service receives an *IBinder* object from an application and transacts with the corresponding Binder server. First, when processing a time-consuming request, the system service can send the intermediate results to a *callback* Binder server asynchronously without interrupting the main request. Second, the system service may need to request data from another Binder server, which can access some system resources not available to the former.

3 BITRE ATTACK FAMILY

Next, we introduce the BiTRe attack family. We start with explaining the threat model and two key observations leading to the discovery of BiTRe, then formally describe BiTRe's workflow, followed by a high-level analysis of its consequences.

3.1 Threat Model

Attacker's Capabilities. The attacker can access all system services and invoke all the interface functions defined in their Binder servers. This can be achieved by either installing a malicious app or compromising an intermediate process. We assume the app/process has the required permissions to access these system services. For example, when accessing the *ICamera* interface, the app/process should have *android.permission.CAMERA* permission and in a context that can pass the Android AppOps checks [7] in *CameraService*.

Attacker's Goal. The attacker aims to locally escalate his privilege by communicating with a system service running in another process. Specifically, Local Privilege Escalation (LPE) includes triggering undefined behavior, corrupting memory, leaking sensitive information, and gaining code execution—in or from a process with a different set of capabilities than the attacker process.

3.2 Key Observations

Binder Servers in the Role-Reversal Case are Customizable. As the Binder proxy in the role-reversal case is provided by the application, an application can instantiate a Binder server by itself, and send the proxy to the system service. This scenario is sometimes necessary, thus Android provides some off-the-peg templates to facilitate the application developers in constructing Binder servers. These templates ensure the transmitted data are syntactically correct. Unfortunately, the system service cannot verify that these templates are followed due to the next reason.

Absence of Mutual Authentication Mechanism. Despite the wide adoption of the role-reversal case, Android does not provide a uniform method for a Binder client to authenticate (1) the Binder server's identity, (2) its permission, and (3) how the Binder proxy is implemented. This absence of client-side authentication tools makes securing the Binder client a challenging task. Admittedly, most vulnerabilities could be avoided with careful input validation and logic examination. However, these checks are often ignored [44] as it is common to trust the server in the C/S architecture [4].

3.3 BiTRe Attack Workflow

Combining these two observations, we discover that, in the role-reversal case, an attacker can construct and instantiate an evil Binder server, and send its proxy as an *IBinder* object to the system service. The system service may misperceive this customized Binder server as a trustworthy one, as there is no easily accessible tool to distinguish them. Note that the system service cannot check the permission set of the Binder server, either, as such an operation can be performed only after authentication.

We define two acronyms here that are crucial to this study:

TBI (Target Binder Interface). A Binder interface whose corresponding system service (1) is reachable by an application, either directly via *ServiceManager*, or indirectly as a sub-interface of a reachable system service, and (2) can receive a Binder proxy from an application and act as a Binder client in the role-reversal case.

CBI (Customizable Binder Interface). A Binder interface that can be customized by an attacker and act as a Binder server to transact with a TBI server in the role-reversal case.

When the context is clear, we also use TBI/CBI to denote their corresponding servers. Their member functions are abbreviated as *TBIF* and *CBIF*, respectively. The BiTRe attack workflow, illustrated in Fig. 3, is as follows:

⓪ **Construct the Application and the CBI Class.** The attacker develops a seemingly benign application to transact with the TBI. A special class in the application is derived from the server-side class—whose name starts with *Bn*—of a CBI with all its interface functions overwritten.

① **Get the TBI's Binder Proxy.** The attacker's application, executing as an unprivileged process, acquires the TBI's Binder proxy, either by calling *getService* in *ServiceManager* or by calling a function in a Binder server which can generate a sub-interface as the TBI.

② **Instantiate the Customized CBI.** The attacker's application instantiates the customized CBI and then constructs its Binder proxy.

③ **Set the CBI in the TBI.** The attacker's application interacts with the TBI server, and eventually sends the *IBinder* object corresponding with the CBI server to the TBI server. The interaction is TBI-specific, as the TBI server accepts an *IBinder* object—i.e., sets the CBI—only if the TBI server is in the right state. This requires its TBIFs invoked in the correct order and all the inputs of the invoked TBIFs are syntactically correct.

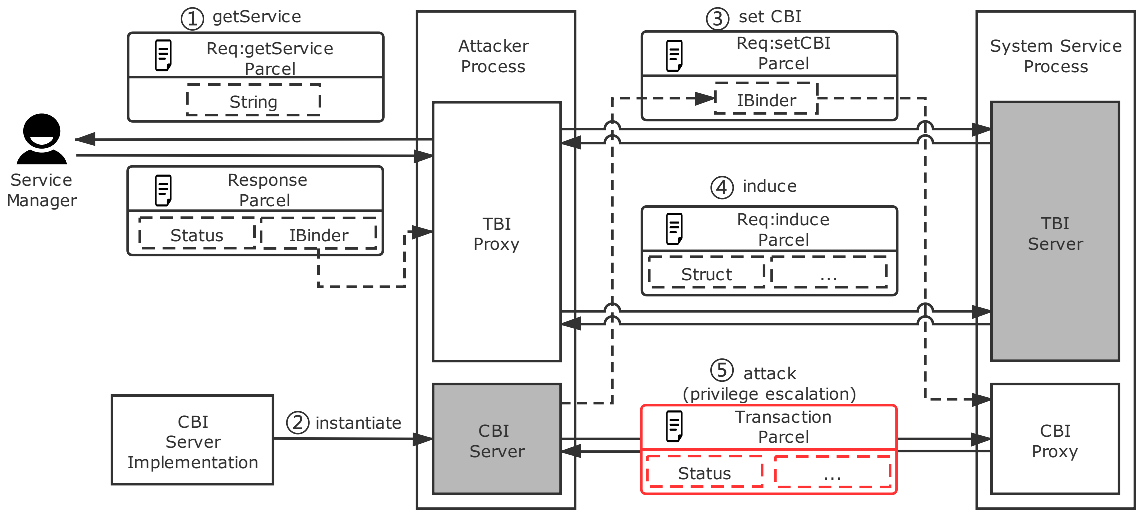

Figure 3: BiTRe attack workflow. The attacker process induces the system service to accept and to communicate with his customized CBI server. The attacker can respond with malformed data to the privileged process via its CBI server as long as the system service transacts with the CBI server. This results in potential privilege escalation attacks.

④ **Induce the TBI Server to Transact with the CBI.** The application calls certain TBIFs to trigger a transaction between the TBI server—acting as a Binder client—and the CBI server. The redirection is successful if the CBI server receives a request.

⑤ **Attack from the CBI.** The CBI server can reply with malformed data to attack the system service, which is often unprotected as the corresponding functions are deemed unreachable by the attackers.

Steps (a) to (c) in Sect. 1 are mapped to steps ③ to ⑤ here.

We craft a buffer overflow bug in an imaginary system service *TService* to illustrate the attack flow. In Line 7 to 10 of *TService*, it queries an array index from the CBI without sanitization:

```
1  class TService : public BnTBI {
2  public:
3    status_t setCBIProxy(sp<CBI> cbi) {
4      this->mCBI = cbi;
5      return OK;
6    }
7    status_t bofBug() {
8      mBuffer[mCBI->getIndex()] = 'c'; // no range check
9      return OK;
10   }
11 private:
12   sp<CBI> mCBI;
13   char mBuffer[];
14 }
```

The attacker can then customize an evil CBI server and implement the corresponding *getIndex()* function (Line 2 to 4 below), returning a malformed index that triggers the vulnerability:

```
1  class EvilCBIServer : public BnCBI { // step 0
2      int getIndex() { // step 5
3          return 0xdeadbeaf;
4      }
5  };
6  int main(){
7      sp<TBI> service =
8          ServiceManager::getService("tservice"); // step 1
9      sp<CBI> cbi = new EvilCBIServer; // step 2
10     service->setCBIProxy(cbi); // step 3
11     service->bofBug();  // step 4
12 }
```

3.4 Attack Consequences

BiTRe opens new attack surfaces previously neglected by both academia and industry. Since many TBI servers execute in privileged processes, such as *system_server* and *media_server*, allowing the attacker to transact with these TBIs as a Binder server results in trust boundary violations. If such violations are not detected, which happens frequently [31], the attacker can escalate its privileges and cause more damage by gaining additional permissions and accessing sensitive system resources.

Specifically, the attacker may induce the system service to send sensitive data to the CBI server, which concern the user's privacy. Or, when the system service performs tasks based on the CBI server's response, the attacker can send malformed data to trigger undefined behaviors if the service is not well protected.

Safeguarding this new attack surface is challenging due to its worrisome scale and its increasing trend, which are revealed next.

4 ATTACK SURFACE ENUMERATION

To quantify the pervasiveness of BiTRe attacks, we first enumerate the potential attack surface—TBIs and CBIs—implemented in C++. We leave Java interfaces, which are also vulnerable to BiTRe attacks but are of different challenges, to future work. Some Binder interfaces' implementation is a combination of Java, C++, and C. Among these Binder interfaces, we include those with C++ Binder proxies in our enumeration.

We start with a high-level description of our data extraction tool and the enumeration process (Sect. 4.1). The detailed process is described from Sect. 4.2 to 4.4. Our results in Sect. 4.5 demonstrate not only the pervasiveness—roughly 32% of Binder interfaces in Android 11 are TBIs and 47% of them are CBIs, but also an increasing trend of the attack surface, as these numbers grow along with Android release versions.

4.1 Data Extraction Method

Analyzing Stage and Analyzing Tools. We choose to scrutinize the system during compilation in its IR form, when all the code is generated, and the class names and the inheritance relationships are still visible. For the tools, we implement several LLVM compiler plugins. LLVM, as Android's default compilation toolchain since Oct 2016 [37], provides powerful APIs for us to comprehensively enumerate the attack surface. By combining these APIs, we can query, in every source file, the list of functions, their argument lists, and the *type* of each argument. A type can either be a *synthetic type*—class or struct—or a *primitive type*, such as integer or char. Synthetic types can be further decomposed via LLVM APIs. Given a function, we can query whether it is a member function of a class, and if so, we can restore the class inheritance chain of the class. Our rationale for the choices and the detailed data extraction mechanisms are in Appendix A.

Enumeration Process. To ensure a thorough enumeration, we insert a snippet of compiler flags in the global configuration script named *global.go*, so that the plugin is executed every time a C++ source file is interpreted into its IR form. Such configuration ensures that LLVM feeds all Android functions, one by one, to our plugin for processing. The processing consists of three steps:

(1) **Binder Interface Recognition.** We determine whether this function is a member function of a Binder interface, and if so, we record the member function and its parameter information for the Binder interface.
(2) **Reachability Analysis.** For each Binder interface, we check whether its proxy can be acquired by an application, either directly replied by *ServiceManager* or sent by other reachable Binder interfaces as a sub-interface.
(3) **Screening for TBIs and CBIs.** For each reachable Binder interface, we check whether it can receive *IBinder* objects by recursively decomposing the parameters in its member functions. A Binder Interface is a TBI if it is reachable, and at least one of its interface functions accepts *IBinder* objects. The same process also gives us the list of CBIs, which are *IBinder* objects that can be transmitted to TBIs.

4.2 Binder Interface Recognition

Our Method. Previous studies [3, 32] recognize Binder interfaces and extract their functions in the (un)marshalling process of a Binder server (⑥ or ⑦ in Fig. 2). However, this method is gradually invalidated by the adoption of *SafeInterface* [5], which hides the details of (un)marshalling from both the developers and the analyzers. Therefore, we alternatively recognize Binder interfaces by analyzing the inheritance relationships, which also guarantees an accurate list of Binder interfaces. The same process also gives us the function parameters of each interface function and the types of these parameters. Compared to previous studies, this approach has an additional benefit of facilitating vulnerability confirmation and further exploitation, as the function names and argument types are preserved, which ensure that the PoRs we generated to confirm the attacks (Sect. 6.1) are (1) syntactically correct, therefore have a high success rate, and (2) human-readable, therefore facilitate the subsequent privilege escalation attacks, whose discovery often requires human involvement.

Fingerprinting Binder Interfaces. As shown in Fig. 1, there are five interfaces/classes associated with a Binder interface: a general interface *IInterface* deriving all C++ Binder interfaces; an interface definition class extending *IInterface*; a Binder proxy class with prefix *Bp*; a Binder native class with prefix *Bn*; the Binder server's actual implementation class extending the *Bn* class. Among these classes, we choose the Binder proxy class with prefix *Bp* as a fingerprint to identify Binder interfaces and hence their member functions, based on the following reasoning.

Taking the classes in Fig. 1 as an example, all the functions defined in the Binder interface *IAudioFlinger* are implemented in its *Bp* classes. Consequently, by verifying whether a function is a member function of a *Bp* class, we learn the exact set of Binder interface functions. In contrast, if a function has a *Bn* class or *IInterface* in its class inheritance chain, it may not be a Binder interface function, as some functions in *AudioFlinger* are not declared in *IAudioFlinger* due to the multiple inheritance feature in C++.

Verifying Binder Interface Functions. We are now ready to describe our Binder interface recognition process. When LLVM feeds our plugin with a function, we first recover its original namespace and class name—which is available if it is a class member function—via a process called *demangling*. The recovered class name is cross-checked with the type name extracted from the function's *this* pointer. If the class name matches the type name, we are certain that this function is a class member function. We check whether a class member function is a Binder proxy function by verifying both of the following conditions: (1) its class inherits *IInterface* indirectly. (2) its class does not implement the *onTransact* function, which is a symbol of *Bn* classes. When both conditions are met, we get the Binder interface name, which is the parent class of the *Bp* class, and all interface functions associated with this Binder interface. After enumerating all Android functions, we now have the complete list of Binder interfaces and their functions.

4.3 Reachability Analysis

A Binder interface serves as a TBI only if its proxy can be acquired by an application. The attacker may get the Binder proxy either by querying *ServiceManager* or by interacting with a reachable Binder server, which returns a sub-interface. The latter case has three caveats: first, sub-interfaces can be nested—i.e., a sub-interface may return another sub-interface; second, not all sub-interfaces are transmitted as return values—some are passed to the application among the out parameters, e.g., those of the TBI *ICameraService* [6]; third, the received sub-interface may be encapsulated in a sub-field, either in a Parcelable object or a pointer whose type is opaque and LLVM fails to extract.

With these caveats in mind, we design our reachability analysis as follows, whose pseudocode is in Appendix D.

Step 1: Querying *ServiceManager*. We execute the *service list* command in a running device, which returns all the Binder interfaces registered in *ServiceManager*. We add them in a set called Reachables.
Step 2: Listing the Return Values and Out Parameters. Our results from Sect. 4.2 allow us to list all the member functions of the newly added elements in Reachables. We then compose a list of these functions' return values and out parameters, named Values. The return values are extracted by querying the function's *sret* attribute via an LLVM API. We manually inspect all these functions to list the out parameters. We observe an interesting fact that could speed up future processing: out parameters are passed to a function by pointer, whereas in parameters are passed either by value or by reference.
Step 3: Decomposing the Return Values and Out Parameters. Now we screen the list Values for *IBinder* objects. All primitive types are dropped. *IBinder* objects are added to Reachables. A non-*IBinder* synthetic type is further decomposed, whose member elements are added to the end of the list for screening.

Dealing with pointers brings another challenge, as a pointer may refer to an *opaque type*, whose definition is in another file. Storing all the definitions of opaque types would significantly downgrade the efficiency of our plugin. To tackle this challenge, we introduce two rounds of pre-processing: the first round records all the opaque type names in all Binder interface functions' arguments and return values; the second round records the definitions of these opaque types. Therefore, when encountering an opaque type in Values, we can look up its definition in our pre-processing results, which are a lot smaller than all the opaque type definitions in Android. The decomposition process continues until Values is empty.

Steps 2 and 3 are executed repeatedly until all elements in Reachables are processed and Values is empty.

4.4 Screening for TBIs and CBIs

After identifying the reachable Binder interfaces, we can now traverse them to get the lists of TBIs and CBIs. We start by composing a list of all the member functions of Binder interfaces in Reachables. For each member function in the list, we decompose all its input parameters to look for attacker-customizable *IBinder* objects. We omit the details as the decomposition process is very similar to Step 3 in our reachability analysis. For each customizable *IBinder* object, we mark the member function as a TBIF, its corresponding Binder interface as a TBI, and the *IBinder* object's corresponding Binder interface as a CBI. The TBIF-CBI connections are also recorded in a *call graph database*.

4.5 Enumeration Results

Statistics in Android 11. The latest Android version is 11 in May 2021, the time of this writing. In release *android-11.0.0_1r*, we count 1465 Binder interface functions in 176 Binder interfaces, among which we identify 57 TBIs and 84 CBIs, whose relationships are visualized in Fig. 4. In these TBIs, 203 member functions accept customizable *IBinder* objects. We call these member functions *corresponding TBIFs (CTBIFs)*, which are crucial for the attacks as they serve as the attacker's entry points. We list the TBIs with the highest numbers of CTBIFs in Appendix C. There are 340 CBIFs reachable from TBIs, which will be analyzed further in Sect. 5.

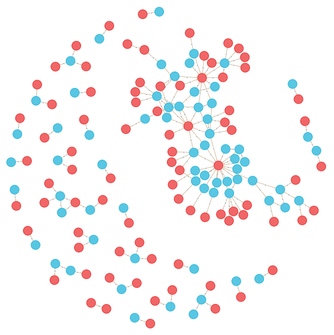
Figure 4: The relationships between TBIs (blue) and CBIs (red) in Android 11. On average, a TBI uses 2.07 CBIs and a CBI can be sent to 1.37 TBIs. A TBI uses at least 1 CBI, at most 7 CBIs.

Table 1: Statistics of recent Android versions' initial stable releases. We use (+29, -16) to denote that there are 29 CTBIFs newly introduced in Android 9, and 16 CTBIFs from Android 8 are deprecated.

Release	TBI	CTBIF	CBI	CBIF
8.0.0_r1	46	125	52	279
9.0.0_r1	47	138 (+29, -16)	51	276
10.0.0_r1	55	178 (+50, -10)	66	320
11.0.0_r1	57	203 (+52, -27)	84	340

Attack Surface Evolution. With the efforts of the Android team, the attack surfaces of most Android components have been reduced and will continue to be reduced [29]. This phenomenon inspires us to analyze how the BiTRe attack surface evolves. The results from Android 8 to 11 in Table 1 exhibit an increasing trend in all the metrics, apart from a slight decrease in CBIs and CBIFs from Android 8 to 9. This decrease results from the introduction of project Treble [10], which migrates some Binder interfaces to the hardware Binder framework, so that hardware-related details are separated from the operating system.

This growing trend is alarming when compared with many other attack surfaces. Unfortunately, it is impractical to hope that the BiTRe attack surface would be reduced in future releases, as Android will grow only more powerful and support more features.

5 ATTACK CAPABILITY ANALYSIS

As CBIs interact directly with the target system service, they are the key components in launching subsequent attacks. The data sent from and received by CBIs translate directly to the number of attack strategies. Specifically, the data received/input from the system service may contain the user's private information, or leak some memory addresses, allowing the attacker to bypass Address Space Layout Randomization and to pinpoint attack locations and *gadgets*—snippets of code in the target program to facilitate the attack. Sending/outputting data to the system service is more damaging as it may directly trigger some undefined behaviors, such as affecting the service's control flow or causing memory corruption.

Given the importance of CBIs' input and output capabilities, in this section, we measure these capabilities via a new metric called *Interface Complexity*, and then rank all CBIs based on this metric, which provides valuable guidance in our later vulnerability discovery.

5.1 Interface Complexity: Definition

Our Interface Complexity metric generalizes the attack surface metric [34] by decomposing the functions to their inputs and outputs and assigning weights according to their data types. To compute Interface Complexity, denoted as C_{IO}, of a CBI b, we first decompose its member functions' inputs and outputs into four data types, whose names are abbreviated from primitive, array, container, and Binder:

Pr. All primitive types with fixed lengths, such as integer, bool, char, and float.

Ar. A known number of fixed-size elements, which can either be of primitive types or fixed-size objects.

Co. A flexible number of fixed-size elements. Typical containers are vectors, maps, and sets.

B. A transactable element referring to another Binder Interface.

We omit our CBI decomposition process as it is identical to that of TBIs (Steps 2 and 3 in Sect. 4.3). We use $n_{\mathrm{I}}^{\mathrm{Pr}}(b)$, $n_{\mathrm{I}}^{\mathrm{Ar}}(b)$, and $n_{\mathrm{I}}^{\mathrm{Co}}(b)$ to denote the numbers of primitive, array, and container items after b's input decomposition, respectively; $n_{\mathrm{O}}^{\mathrm{Pr}}(b)$, $n_{\mathrm{O}}^{\mathrm{Ar}}(b)$, and $n_{\mathrm{O}}^{\mathrm{Co}}(b)$ denote the corresponding counts in b's output decomposition. The set of Binder interfaces are represented by $B_{\mathrm{I}}(b)$ and $B_{\mathrm{O}}(b)$. Then we assign a weight value to each decomposed item of the first three types, based on (1) whether it is decomposed from an input parameter or an output, and (2) its data type. An intermediate result $C_{\mathrm{IO}}^{\mathrm{Pr,Ar,Co}}(b)$ is calculated as the total weight of the first three types:

$$\begin{aligned} C_{\mathrm{IO}}^{\mathrm{Pr,Ar,Co}}(b) &= w_{\mathrm{I}}^{\mathrm{Pr}} \cdot n_{\mathrm{I}}^{\mathrm{Pr}}(b) + w_{\mathrm{I}}^{\mathrm{Ar}} \cdot n_{\mathrm{I}}^{\mathrm{Ar}}(b) + w_{\mathrm{I}}^{\mathrm{Co}} \cdot n_{\mathrm{I}}^{\mathrm{Co}}(b) \\ &\quad + w_{\mathrm{O}}^{\mathrm{Pr}} \cdot n_{\mathrm{O}}^{\mathrm{Pr}}(b) + w_{\mathrm{O}}^{\mathrm{Ar}} \cdot n_{\mathrm{O}}^{\mathrm{Ar}}(b) + w_{\mathrm{O}}^{\mathrm{Co}} \cdot n_{\mathrm{O}}^{\mathrm{Co}}(b) \ . \end{aligned}$$

The actual Interface Complexity is calculated by adding $C_{\mathrm{IO}}^{\mathrm{Pr,Ar,Co}}(b)$ with the discounted sum of all decomposed Binder items in its inputs and outputs:

$$\begin{aligned} C_{\mathrm{IO}}(b) = C_{\mathrm{IO}}^{\mathrm{Pr,Ar,Co}}(b) &+ \alpha_{\mathrm{I}} \sum_{b' \in B_{\mathrm{I}}(b)} C_{\mathrm{IO}}^{\mathrm{Pr,Ar,Co}}(b') \\ &+ \alpha_{\mathrm{O}} \sum_{b' \in B_{\mathrm{O}}(b)} C_{\mathrm{IO}}^{\mathrm{Pr,Ar,Co}}(b') \ , \end{aligned}$$

where α_{I} and α_{O} are two discount factors. The last step can be iterated to incorporate the Binder items referred to by members of $B_{\mathrm{I}}(b)$ and $B_{\mathrm{O}}(b)$. However, such extra complexity does not contribute to the accuracy in our case.

As an initial attempt, we assign weights 1, 2, 2 to the first three data types, regardless of whether the item is from the inputs or the outputs, and choose $\alpha_{\mathrm{I}} = \alpha_{\mathrm{O}} = 1$. This simple Interface Complexity definition already manifests its usefulness in our later attack confirmation.

5.2 Interface Complexity: Results

CBIs with High $C_{\mathrm{IO}}(b)$ and Type I Attacks. We calculated the Interface Complexity of all CBIs in Android 11, and selectively list some results in Table 2. Most vulnerabilities we discovered—20 out of 26—reside in CBIs with the highest Interface Complexity, proving a strong correlation between Interface Complexity and attack capability. These 20 vulnerabilities cover all Type I attacks in Sect. 6.2, which include four CVEs out of the ten CVEs corresponding to this study. We release the complete Interface Complexity results[1] to help Android developers and security researchers prioritize their efforts.

CBIs with Non-Empty $B_{\mathrm{O}}(\cdot)$ & $B_{\mathrm{I}}(\cdot)$ and Type II & III Attacks. By decomposing a CBI b, we get the set of CBIs that b can send to TBIs, i.e., $B_{\mathrm{O}}(b)$, and the set of TBIs b can receive, i.e., $B_{\mathrm{I}}(b)$. We call members of $B_{\mathrm{O}}(b)$ *sub-CBIs* and members of $B_{\mathrm{I}}(b)$ *sub-TBIs*. We pay special attention to these sets in our vulnerability discovery, as these attack portals involving multiple CBIs or TBIs are stealthier than the others. We use *Type II attacks* to denote vulnerabilities involving sub-CBIs, and *Type III* for sub-TBIs. Of these two types, we found six vulnerabilities that lead to six CVEs (cf. Sect. 6.2).

6 CONFIRMING THE BITRE ATTACKS

A complete BiTRe attack involves two phases: *preparation phase* (Steps ⓪ to ④ in Sect. 3.3), during which the attacker process transacts with the TBI and induces it to call a CBIF as a Binder client, and *attacking phase* (Step ⑤), during which the CBI launches various attacks. We explore these two phases separately.

For the preparation phase, as the attacker can almost always trick a TBI server into transacting with the evil CBI by mimicking a benign application, there is no need to analyze its feasibility. Therefore, we turn to demonstrate the attacks' efficiency by building an automatic system, which generates PoR programs to transact with TBIs, hoping to induce them to call CBIFs without any knowledge of the Binder interfaces' semantics.

For the attacking phase, we manually review the source code and perform dynamic tests on more than half of the CBIs, starting from those with the highest Interface Complexity. We find 26 vulnerabilities listed in Table 5 in Appendix E with different root causes and crash points, categorize them into three types, and introduce the characteristics of these types.

6.1 Preparation Phase: PoR Generation

Overview. Our PoRs do not trigger or exploit vulnerabilities; we consider a PoR successful if a role-reversal transaction involving a new TBI-CBIF pair is initiated. We leave the fully automatic discovery of BiTRe vulnerabilities to future work. Our system consists of two components: a *PoR generator* and a *PoR executor*. The former generates executable PoR programs targeting each TBI-CBIF pair, and feeds them to the PoR executor, which executes the PoR in an Android device. The process repeats until the CBIF is invoked, at which time the PoR is recorded. Each PoR is a combination of a plausible TBIF calling sequence and a set of plausible inputs to these TBIFs. The success of a PoR relies on generating the correct calling sequence and the inputs, which is explained next.

[1]https://github.com/xiangxiaobo/BiTRe

Table 2: CBIs with the highest Interface Complexity. Those where we discover vulnerabilities are in bold, with the number of vulnerabilities in parentheses. CBIs with rankings are from Android 11; *IGraphicBufferConsumer*'s vulnerabilities are from Android 7—it is no longer a CBI in Android 11. INPUT_CAP and OUTPUT_CAP are the data types involved in the CBI's input parameters and outputs, respectively.

Rank	CBI Name	Interface Complexity $C_{\mathrm{IO}}(\cdot)$	INPUT_CAP	OUTPUT_CAP
1	IMediaPlayer	875	{Pr, Ar, Co, B}	{Pr, Ar}
2	IRemoteDisplayClient	506	{Pr, Ar, B}	{Pr}
3	**IGraphicBufferProducer (11)**	492	{Pr, Ar, Co, B}	{Pr, Ar}
*	**IGraphicBufferConsumer (9)**	432	{Pr, Ar, Co, B}	{Pr}
4	IIdentityCredential	382	{Pr, Ar, Co}	{Pr, Ar}
5	ICredential	282	{Pr, Ar, Co}	{Pr, Ar}
...	...	...	...	...
30	**IDataSource (4)**	31	{Pr}	{Pr, Ar, B}
31	**IPullAtomCallback (2)**	28	{Pr, Ar, B}	{Ar}
...	...	...	...	...

Calling Sequence Generation. To trigger a CBIF, we first locate its CTBIF (TBIF that calls the CBIF) by searching the call graph database established in Sect. 4.4. If we are lucky, a direct invocation of the CTBIF completes the job already. However, invoking the CBIF often requires the TBI server to be in a specific state, which further requires the application to invoke the correct sequence of TBIFs. Unfortunately, it is infeasible to directly extract such a sequence, as it requires a deep understanding of the TBI's semantics. We observe, after reviewing the documents, that such a calling sequence usually involves only a small number of TBIFs. Therefore we adopt a strategy inspired by fuzz testing, which randomly calls some TBIFs before invoking the CTBIF, hoping to set the TBI server in the correct state. This simple strategy proves effective.

Input Generation. Our Binder interface recognition (Sect. 4.2) gave us the syntax information of the TBIF inputs already. We can therefore utilize this information to generate all the primitive type inputs and to mutate them during fuzzing.

The remaining challenge is to generate semantically correct synthetic-type inputs—usually class or structure objects. We first search for existing TBIFs that output the object. Their outputs are often semantically correct as these functions are provided by the system. When such functions are not available, we prefer the object's constructor, as it often presets legal default values of its members. If the constructor cannot provide a valid object either, our system constructs the object from scratch. We developed a Clang plugin during building the AOSP to extract the type information along with each member's name in each class/structure. With the type information, we decompose the object one layer at a time and try the aforementioned methods to construct its synthetic type elements, rather than decomposing the object directly to primitive type items as in Sect. 4.3. We also manually prepare 15 generators for some types that are particularly difficult to construct, e.g., those with magic number fields that must be set to specific values.

Executing the Prototype System. We execute our PoR generator in a Ubuntu 20.04 system with an Intel Core i7-6700 @ 3.40GHz processor and 32G RAM. The PoR executor runs on a Google Pixel 3 device with the build tag android-11.0.0_r1.

During the process, we discover another method to affect the TBI's control flow from a CBI. Specifically, some TBIs create a *death recipient* for their corresponding CBIs, so that they can conduct garbage collection once the CBI dies. In other words, the CBI affects the TBI's state and control flow without a CBIF invocation.

In 24 hours, our system triggers 57 CBIFs and 24 death recipients, involving 28 CBIs and 12 TBIs. Selected results are in Appendix B. The attack portals corresponding to seven CVEs are reachable from our PoRs, including three Type I and four Type II attacks.

6.2 Attacking Phase: Triggering Vulnerabilities

We find and report to Google 26 vulnerabilities in total, each with its distinct root cause and crash point. Ten of them are assigned CVEs; most of the others are marked as duplicates with these ten. We do not claim to exhaust the attack family as we have not covered all the TBIs and CBIs, but only the ones with high Interface Complexity, and those involve sub-CBIs and sub-TBIs (cf. Sect. 5.2). All 26 are fixed in subsequent Android releases based on our reports. These vulnerabilities involve four "TBI ⇒ CBI" pairs:

- *ISurfaceComposer* ⇒ *IGraphicBufferProducer* with 11 cases;
- *IOMX* ⇒ *IGraphicBufferConsumer* with nine cases;
- *IMediaExtractor* ⇒ *IDataSource* with four cases;
- *IStatsd* ⇒ *IPullAtomCallback* with two cases.

We split these vulnerabilities into four types based on the number of CBIs and TBIs involved and illustrate them in Fig. 5. Type I attacks involve only two Binder interfaces: a TBI and a CBI. In Type II attacks, one TBI and multiple CBIs are involved. The sub-CBI causes the actual damage. In Type III attacks, one CBI and multiple TBIs are involved. The attacker leverages the evil CBI to attack the sub-TBI. In Type IV attacks, multiple TBIs and multiple CBIs are involved. Sub-CBIs are utilized to attack certain sub-TBIs.

Type I: Direct Attack. Type I is the basic pattern of BiTRe attack, on which the other three types are based. The attacker constructs evil CBI Servers and sends malformed data to the TBI, triggering insecure data- and control-flow vulnerabilities in the TBI.

The workflow of Type I attacks is shown at the end of Sect. 3.3. All 20 vulnerabilities involving TBIs *ISurfaceComposer* and *IOMX* belong to this type.

A typical example is CVE-2017-0665, where the attacker customizes the CBI *IGraphicBufferProducer* to interact with the TBI

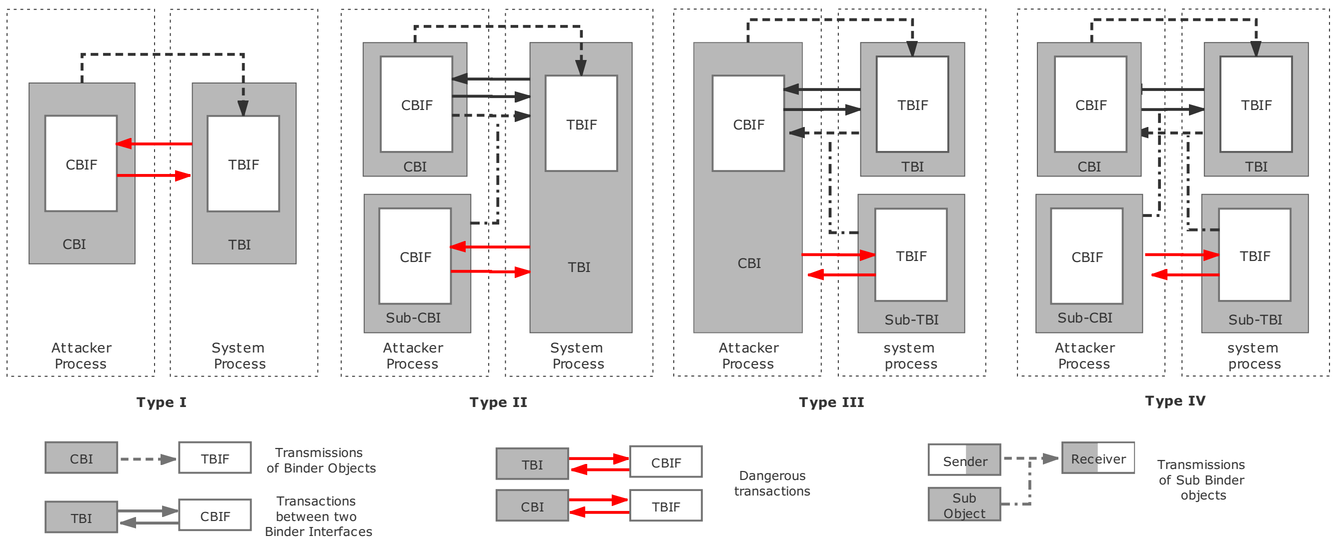

Figure 5: Four types of BiTRe attacks. In Type I attacks, the CBI sends malformed data to reply to the TBI's requests. In Type II attacks, the CBI to launch the attack, i.e., the sub-CBI, is sent to the TBI by the upper CBI. In Type III attacks, the TBI under attack, i.e., the sub-TBI, is sent to the CBI by the upper TBI. In Type IV attacks, multiple CBIs and TBIs are involved.

ISurfaceComposer. We successfully exploit this vulnerability to hijack the control flow of *SurfaceFlinger*, which is the actual implementation of the TBI Server. When an application calls *BpSurfaceComposer::captureScreen()* of *SurfaceFlinger* with an evilly customized CBI Server *IGraphicBufferProducer*, *SurfaceFlinger* does not sanitize the return values of the *dequeueBuffer()* implemented in CBI *IGraphicBufferProducer*. The replied position is used to access an array named *mSlot*, leading to out-of-bounds access issue. The out-of-bounds memory is then converted to an *ANativeWindowBuffer* object which contains two function pointers. In function *eglCreateImageKHR* that is invoked indirectly by the *BpSurfaceComposer::captureScreen()* , there is a *blx r2* instruction. The *r2* register is read from the buffer after being dequeued. As we carefully spray the heap, we can overwrite the register and eventually hijack the control flow of the *SurfaceFlinger* context.

Type II: One TBI, Multiple CBIs. Many Binder interfaces have functions that return another Binder object. Whenever these Binder interfaces act as CBIs, we can construct a malicious sub-CBI server and send it to the TBI via these functions. Compared to Type I attacks, Type II vulnerabilities are more often ignored by the developers as the attack portal is nested deeply in the code structure.

We extend *TService* in Sect. 3.3 to illustrate Type II attacks. In Line 7 to 9 below, *setSubCBI* function requests another CBI called *mSubCBI* from the CBI:

```
1  class TService : public BnTBI {
2  public:
3    status_t setCBIProxy(sp<CBI> cbi) {
4      this->mCBI = cbi;
5      return OK;
6    }
7    status_t setSubCBI(){
8      mSubCBI = mCBI->getSubCBI();
9    }
10   status_t bofBug(){
11     mBuffer[mSubCBI->getIndex()] = 'c'; // no range check
12   }
13 private:
14   sp<SubCBI> mSubCBI;
15   sp<CBI> mCBI;
16   char mBuffer;
17 }
```

The attacker then implements two Binder servers—*EvilCBIServer* in Line 1 to 5 below for providing the sub-CBI and *EvilSubCBIServer* in Line 6 to 11 for returning the malformed index:

```
1  class EvilSubCBIServer : public BnSubCBI {
2      int getIndex(){
3          return 0xdeadbeaf;
4      }
5  };
6  class EvilCBIServer : public BnCBI {
7      sp<SubCBI> getSubCBI(){
8          sp<SubCBI> subCBI = new EvilSubCBIServer;
9          return subCBI;
10     }
11 };
12 int main() {
13   sp<EvilCBIServer> cbi = new EvilCBIServer;
14   sp<TBI> service = ServiceManager::getService("tservice"
       );
15   service->setCBIProxy(cbi); // set evil cbi
16   service->setSubCBI(); // set sub-CBI
17   service->bofBug();  // trigger the bof bug.
18 }
```

The four vulnerabilities involving the TBI *IMediaExtractorService* are of this type, with four CVEs assigned. In these cases, *IMediaExtractorService* requests an *IMemory* proxy from our CBI *IDataSource*. The actual attack is launched from the sub-CBI *IMemory*. Specifically, *IMediaExtractorService* requests the same data multiple times from *IMemory*, which allows the attacker to reply with inconsistent data and thus cause a heap-based buffer overflow. The four vulnerabilities differ in the requested data fields.

Type III: One CBI, Multiple TBIs. When a CBIF has a Binder object among its input parameters, the corresponding TBI must construct and send the Binder object to invoke the CBIF. The newly constructed Binder object has the same privileges as its constructor TBI, thus attacking it is equally rewarding. However, it is neither registered in *ServiceManager*, nor considered as a sub-interface (sub-interfaces are directly sent by TBI to the application, not via a CBI), therefore more likely to be overlooked by the developers as an attack target. Moreover, unlike the previous two types where the CBI passively waits for the TBI server's invocations, in Type III attacks, the attacker can initiate function calls as it has received the Binder proxy, therefore enables a comprehensive vulnerability exploration of the Binder object.

We modify *TService* again to illustrate these attacks. In Line 14 to 18 below, the TBI declares a *sendSubTBI* function that initializes another Binder server and sends it to the CBI as a parameter of the CBIF *callback*. The buffer overflow bug is in Line 2 to 4 of *SubTBI*.

```
1  class SubService: public BnSubTBI{
2      void bofBug(int index){
3          mBuffer[index] = 'c'; // no range check
4      }
5  private:
6    char mBuffer[];
7  }
8  class TService : public BnTBI {
9  public:
10   status_t setCBIProxy(sp<CBI> cbi) {
11     this->mCBI = cbi;
12     return OK;
13   }
14   status_t sendSubTBI(){
15       sp<SubTBI> subTBI = new subTBI;
16       mCBI->callback(subTBI);
17       return OK;
18   }
19 private:
20   sp<CBI> mCBI;
21 }
```

The attacker then implements an evil CBI server with *callback* function to receive the Binder proxy of *SubTBI* (Line 2 below) and to trigger the bug (Line 3).

```
1  class EvilCBIServer : public BnCBI {
2      status_t callback(sp<SubTBI> subTBI){
3          subTBI->bofBug(0xdeadbeaf);
4      }
5  };
6  int main() {
7    sp<CBI> cbi = new EvilCBIServer;
8    sp<TBI> service = ServiceManager::getService("tservice"
       );
9    service->setCBIProxy(cbi); // set evil cbi
10   service->sendSubTBI(); // trigger the bof bug
11 }
```

We found two Type III vulnerabilities in the TBI *IStatsd*. This TBI, once induced to communicate with our CBI *IPullAtomCallback*, constructs and sends a Binder proxy of *IPullAtomResultReceiver*, whose Binder server executes in the same process with the TBI server and enjoys the same privilege. Through invoking *IPullAtomResultReceiver*'s function with malformed buffer, we can trigger two memory corruption bugs via out-of-bounds writes.

Type IV: Multiple CBIs, Multiple TBIs. Type IV is a combination of Type II and III. The attacked TBI is provided by another TBI, and the CBI causing actual damage is from another CBI crafted by the attacker. We have not found any vulnerabilities of this type, but these attack scenarios do exist in the hierarchical Binder interfaces [32] in Android system.

7 DISCUSSION

7.1 Impact of the BiTRe Attacks

Attacking Unprotected Territories. The vulnerabilities we discovered in Binder exceed those in previous studies [25, 28, 42] not only in quantity but also in damage, because we explore a new and pervasive attack surface mostly neglected by both academia and industry. Specifically, existing studies do not attend to the possibility that a Binder server returns malformed data to a system service. Moreover, Binder objects nested among the input parameters and outputs of other Binder interfaces—those involved in Type II and Type III vulnerabilities—are excluded from these studies, as they are deemed unreachable by the attacker.

Non-Triviality of Fixing BiTRe Attacks. There is no silver bullet that can eliminate the BiTRe attacks. The role-reversal case cannot be forbidden as it enables the separation of system services into different Binder servers so that their operations are mostly independent of each other, which is indispensable for Android's security. Enabling universal mutual authentication may results in information leakage from the system services to the applications, which would invalidate, rather than improve, the security model of the C/S architecture.

7.2 Mitigating the BiTRe Attacks

Given the impossibility of a simple defense, we suggest the following mitigation efforts on three different levels.

New Protection Mechanism. Although mutual authentication is inadvisable, we observe that a Binder client can verify whether two Binder proxies are referring to the same Binder server, which leads to our following solution. We suggest introducing a new system service, registered in *ServiceManager* and in charge of constructing Binder servers—corresponding to those in our CBI list—for applications. This service implements strict restrictions to forbid overwriting critical functions. Applications should always commission this service to construct their Binder servers. Note that we cannot prevent an application from bypassing this requirement and constructing CBI servers by themselves; however, that should not be a problem if the next rule is enforced. Whenever a system service first transacts with a Binder server, it should query the Binder-server-construction service if the Binder server is not constructed by itself. The construction service then traverses its global object map and responds with whether the object is constructed by it. The system service aborts the transaction if a match is not found. To avoid information leakage from the Binder-server-construction service, an application, which runs in unprivileged processes, cannot query for Binder servers not commissioned by itself.

Sanitization Measures. Input validation must be enforced in all system services, regardless of whether the corresponding Binder servers/clients are trusted. However, manual sanitization during development is not adequate for ruling out all vulnerabilities. Auxiliary frameworks, such as the RLBox API [36], can be considered for mitigating unknown vulnerabilities.

Raising Awareness. BiTRe attacks should be explicitly addressed in both Android's development and its security audits. Specifically, Android developers should refactor the Binder interfaces to avoid the transmission of active objects whenever possible, thus reducing the role-reversal cases. Security auditors should test all Binder interface functions, rather than excluding those believed to be unreachable by attackers, prioritizing CBIs with high Interface Complexity and their corresponding TBIs.

7.3 Exploring the Folded Attack Surfaces

The vulnerabilities we discovered are by no means exhaustive in the BiTRe family. Moreover, BiTRe is just one example of a greater family, where the attack surface is folded in exceptions (e.g., role reversal enabled by the active object transmission of Binder) of the designer-envisioned security model (e.g., the fixed-role C/S architecture). We suggest future research to explore these folded attack surfaces in the following directions:

Thorough Inspection of the BiTRe Attacks. Our inspection can be extended in two aspects. First, to continue reviewing TBIs and CBIs, both manually and with the help of automatic vulnerability discovery methods such as fuzzing. Second, to include Java system services and hardware/vendor services in the analysis.

BiTRe Attacks in Other Platforms. The temporary reversal of C/S roles is not unique to Android. For example, *svchost*, a Windows system service that hosts many Windows services, can also temporarily act as a client process to interact with a customized server received from an application [35]. The susceptibility of other systems to BiTRe attacks is worth further investigation.

Other Folded Attack Surfaces. The BiTRe attacks resemble the tip of the iceberg, where attackers exploit the inconspicuous exceptions that violate the system's general security model to invalidate its security guarantees. Similar examples include the exceptions of sending *allowed objects* to Android Sandbox against the general model of isolation; the possibility to develop applications in C/C++ in Android against the general adaptation of Java. Nevertheless, these exceptions are usually ignored by both developers and researchers, whose investigations are often restrained by the vision of the system designer—even though the attackers are not. With this study, we, therefore, aim to raise the community's awareness of these folded attack surfaces so that we can work together to unfold their mechanisms, enumerate the vulnerabilities and take precautions.

8 RELATED WORK

We first introduce three types of prior studies on Android IPC security that assumes a fixed Binder-client role of the attacker. Afterward, we highlight two other attacks that also construct Binder servers, which inspire our work.

Permission Related Vulnerabilities. Several studies [2, 12–14, 19] analyze the necessary permissions, which are sometimes not documented, to launch each developer API method. Based on these API-to-permission-set mappings, other studies, including AceDroid [1], ACMiner [22] and Kratos [39], discover that different paths visiting the same system resource may demand different sets of permissions. Further studies [18, 23] extend these studies to visit system resources indirectly via some vulnerable *deputy APIs*, bypassing the permission checks.

Parcel Deserialization Vulnerabilities. The XBRF risk proposed by Rosa [38] cannot cause any damage in practice, as the attack is defended by Binder's object searching algorithm. XBRF is not similar to BiTRe: it focuses on the deserialization process and does not involve the customization of Binder servers. Another family of Parcelable mismatch bugs, first discovered by Bednarski [15] and then caught in Android Trojans in the wild [40], allow attackers to launch arbitrary components.

Input Validation Vulnerabilities. Input validation in Android is often "unstructured, ill-defined and fragmented" compared to permission checks, making it more challenging to analyze [44]. Researchers conduct code review [21], or builds systems leveraging fuzz testing [16, 20, 27, 32, 32, 43], taint analysis [42], machine learning [44] and symbolic execution [33] to find vulnerabilities in system services. For countermeasures, Android fixes numerous vulnerabilities [31] and applies multiple exploit containment measures [29] to restrain the security impact even when some processes are compromised.

Vulnerabilities Involving Customized Binder Servers. Wang et al. [42] proposed the "call me back" attack via constructing irresponsive callback functions and resulting in a denial of service of a few system services and apps. As a case study among a series of Parcel deserialization bugs [24], He identified a vulnerability in the unmarshalling of *AMessage* objects. Triggering this vulnerability involves constructing a CBI Server, i.e., the *IStreamSource* server, and sending the malformed *AMessage* from *IStreamSource* to *IMediaPlayer*.

9 CONCLUSION

As Binder follows the classic C/S architecture, previous security analyses and studies were trapped by its fixed-role assumption, neglecting the fact that the role-reversal case is widely adopted by both system services and applications. In this paper, we highlighted the severe security implications of this role-reversal case: it allows the attacker to reach a large attack surface that was previously deemed unreachable, thus often unprotected. We discovered a series of vulnerabilities that exploit the attack surface from three approaches. Unfortunately, the vulnerabilities we discovered are by no means exhaustive in the BiTRe family. Neither can the countermeasures we proposed guarantee the elimination of the attack surface. We advocate developers and researchers scrutinize this folded attack surface, among others of similar nature, before they are maliciously exploited by attackers.

ACKNOWLEDGEMENTS

We would like to thank Prof. Kaiyu Shao from China University of Political Science and Law and the anonymous reviewers for their constructive comments. This work is supported by the Strategic Priority Research Program of Chinese Academy of Sciences (No. XDC02040100). Ren is partially supported by Shandong Key Research and Development Program (Grant No. 2020ZLYS09).

REFERENCES

[1] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *NDSS*.

[2] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1164.

[3] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. 2021. Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*.

[4] Muath Alkhalaf, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso, and Christopher Kruegel. 2012. Viewpoints: differential string analysis for discovering client-and server-side input validation inconsistencies. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 56–66.

[5] Android Code Search. 2021. libbinder: Add SafeInterface. https://cs.android.com/android/_/android/platform/frameworks/native/+/d630e520de9ff4bc50723a7e8f91b6d9be27db1c. Accessed on Jan 31, 2021.

[6] Android Code Search. 2021. Source code of CameraService.h in AOSP. https://cs.android.com/android/platform/superproject/+/master:frameworks/av/services/camera/libcameraservice/CameraService.h?q=cameraservice. Accessed on May 7, 2021.

[7] Android developers. 2021. Android AppOpsManager. https://developer.android.com/reference/android/app/AppOpsManager. Accessed on Aug 3, 2021.

[8] Android developers. 2021. Android PermissionChecker Developer API. https://developer.android.com/reference/androidx/core/content/PermissionChecker. Accessed on July 29, 2021.

[9] Android Developers. 2021. Parcel. https://developer.android.com/reference/android/os/Parcel#active-objects. Accessed on Feb 2, 2021.

[10] Android Developers Blog. 2017. Here comes Treble: A modular base for Android. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html. Accessed on Feb 2, 2021.

[11] Android Open Source Project. 2021. Android Interface Definition Language (AIDL). https://developer.android.com/guide/components/aidl. Accessed on Jan 31, 2021.

[12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *the ACM Conference on Computer and Communications Security*. 217–228.

[13] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On demystifying the Android application framework: Re-visiting Android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*. 1101–1118.

[14] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. 2014. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android. *IEEE Transactions on Software Engineering* 40, 6 (2014), 617–632.

[15] Michal Bednarski. 2017. Reparcel Bug. https://github.com/michalbednarski/ReparcelBug. Accessed on Feb 3, 2021.

[16] Cao Chen, Gao Neng, Liu Peng, and Xiang Ji. 2015. Towards Analyzing the Input Validation Vulnerabilities Associated with Android System Services. In *Proceedings of the 31st Annual Computer Security Applications Conference*. Association for Computing Machinery, 361–370.

[17] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. 2017. Analysis of SEAndroid policies: combining MAC and DAC in Android. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 553–565.

[18] William Enck. 2020. Analysis of access control enforcement in Android. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*. 117–118.

[19] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. 627–638.

[20] Huan Feng and Kang G Shin. 2016. BinderCracker: Assessing the Robustness of Android System Services. *arXiv preprint arXiv:1604.06964* (2016).

[21] Guang Gong. 2015. Fuzzing android system services by binder call to escalate privilege. *BlackHat USA* (2015).

[22] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 25–36.

[23] Sigmund Albert Gorski III and William Enck. 2019. ARF: identifying re-delegation vulnerabilities in Android system services. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. 151–161.

[24] Qidan He. 2016. Hey your Parcel Looks Bad, Fuzzing and Exploiting Parcelization vulnerabilities in Android. In *BlackHat Asia, 2016*.

[25] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. 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*. 1236–1247.

[26] Jim Huang. 2012. Android IPC Mechanism. https://www.slideshare.net/jserv/android-ipc-mechanism.

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

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

[29] Nick Kralevich. 2017. Honey, I Shrunk the Attack Surface – Adventures in Android Security Hardening.

[30] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

[31] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. 2017. An empirical study on Android-related vulnerabilities. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2–13.

[32] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. 2020. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In *29th USENIX Security Symposium (USENIX Security)*.

[33] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System service call-oriented symbolic execution of Android framework with applications to vulnerability discovery and exploit generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 225–238.

[34] Pratyusa K Manadhata and Jeannette M Wing. 2010. An attack surface metric. *IEEE Transactions on Software Engineering* 37, 3 (2010), 371–386.

[35] Microsoft Security Update Guide. 2020. CVE-2020-1393 Windows Diagnostics Hub Elevation of Privilege Vulnerability. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2020-1393. Accessed on Jan 31, 2021.

[36] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting fine grain isolation in the Firefox renderer. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 699–716.

[37] Stephen Hines Nick Desaulniers, Greg Hackmann. 2021. Compiling Android userspace and Linux Kernel with LLVM. https://llvm.org/devmtg/2017-10/slides/Hines-CompilingAndroidKeynote.pdf. Accessed on Jan 31, 2021.

[38] Tomáš Rosa. 2011. Android Binder Security Note: On Passing Binder Through Another Binder. https://crypto.hyperlink.cz/files/xbinder.pdf. Accessed on Feb 2, 2021.

[39] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework.. In *NDSS*.

[40] SUDONULL. 2019. EvilParcel Vulnerability Analysis. https://sudonull.com/post/26295-EvilParcel-Vulnerability-Analysis-Doctor-Web-Blog. Accessed on Feb 3, 2021.

[41] Tuna. 2021. Monthly tarball of AOSP. https://mirrors.tuna.tsinghua.edu.cn/aosp-monthly/. Accessed on May 6, 2021.

[42] Kai Wang, Yuqing Zhang, and Peng Liu. 2016. Call Me Back! Attacks on System Server and System Apps in Android Through Synchronous Callback. In *ACM SIGSAC Conference on Computer and Communications Security*. 92–103.

[43] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang. 2017. Exception beyond Exception: Crashing Android System by Trapping in "Uncaught Exception". In *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 283–292.

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

[45] Walter Zimmer. 1995. Relationships between design patterns. *Pattern languages of program design* 57 (1995), 345–364.

A CHOOSING THE ANALYZING STAGE AND THE ANALYZING TOOLS

Analyzing Stage. The intimidating size of Android's source code—115.3 GB as of May 2021 [41]—prevents any manual data extraction approach. An automatic analysis demands us to first choose among the three stages of the system—the source code, the binaries, or analyzing during compilation. Parsing the source code cannot give us a complete list of Binder interfaces as some Binder interfaces are

Table 3: CBIs triggered by our PoRs and their corresponding TBIs. The column "# CBIF" is the number of invoked CBIFs by the TBI. Note that the "IBinder" does not stand for an actual CBI; the "# CBIF" column of "IBinder" is the number of death recipients triggered once killing the CBI server.

TBI	⇒	CBI	# CBIF
IAudioPolicyService	⇒	IAudioPolicyServiceClient	2
	⇒	IBinder	1
ICameraService	⇒	ICameraDeviceCallbacks	2
	⇒	ICameraServiceListener	1
IMediaMetadataRetriever	⇒	IMediaHTTPService	1
	⇒	IDataSource	3
IMediaExtractor	⇒	IDataSource	3
ISurfaceComposer	⇒	IGraphicBufferProducer	3
IAudioFlinger	⇒	IAudioFlingerClient	1
	⇒	IBinder	1
IInputFlinger	⇒	ISetInputWindowsListener	1
...	⇒	...	...

defined and implemented in code generated during building [11]. The extensive use of macros, which are not processed until compilation, also renders a pre-compilation analysis inaccurate. Extracting Binder interfaces from the binary files is also infeasible, as all the names and data structures are lost after compilation, thwarting us from identifying Binder interfaces and active objects. Therefore, we choose to scrutinize the system during compilation in its IR form, when all the code is generated, and the class names and the inheritance relationships are still visible.

Analyzing Tools. As Android's default compilation toolchain since Oct 2016 [37], LLVM provides powerful APIs for us to implement a compiler plugin and thus comprehensively enumerate the attack surface. Here we briefly introduce the capabilities of some LLVM APIs. Each source file is compiled as a *Module* in LLVM. For each Module, we can query its list of functions and their argument lists. For each argument of a function, we can query its *type*, which could either be a *synthetic type*—class or struct—or a *primitive type*, such as integer or char. Synthetic types can be further decomposed. Moreover, we can restore the class name of a class member function by querying the type of *this* pointer—the first or second argument in the member function's IR form. Given a class name, we can query the name of its parent class. By querying the parent class name recursively, we can restore a class inheritance chain.

B LIST OF CBI-TBI PAIR TRIGGERED BY PORS.

The selected list of TBI-CBIF paths that can be triggered by our PoRs is shown in Table 3.

C LIST OF TBIS AND THEIR TBIF & CTBIF COUNTS.

The selected list of TBIs is shown in Table 4. The number of CTBIFs indicates how many entry points to which an attacker can send their CBIs; the number of TBIF shows how many functions which an attacker can leverage to induce the TBI into proper states.

Table 4: TBIs with the highest numbers of CTBIFs in Android release 11.0.0_r1.

Interface name	CTBIF	TBIF
ISurfaceComposer	32	50
IKeystoreService	15	37
IVold	7	81
ICameraService	7	18
IServiceManager	7	9
ICameraDeviceUser	6	23
IWificond	6	14
IMediaPlayer	5	42
IMountService	5	27
ICamera	5	26
IStatsd	5	26
IThermalService	5	11
ISurfaceComposerClient	5	5
...	...	...

Algorithm 1 Searching for IBinder Object in a Param

```
INPUT: $param$                        ▷ a parameter of a Binder function
OUTPUT: $res$                         ▷ a set of IBinder objects in param
1: function SEARCH($param$)
2:     $res \leftarrow \{\}$
3:     for all $type \in param.subtypes()$ do
4:         while $type.isPointerTy()$ do
5:             $type \leftarrow type.getElementTy()$
6:         if $type \in s\_visited$ then
7:             continue
8:         $s\_visited.add(type)$
9:         if $type.name ==$ "IBinder" then
10:            $res.add(subtype)$
11:        else
12:            for all $subtype \in type.subtypes()$ do
13:                $res.union(\text{SEARCH}(subtype))$
    return $res$
```

Algorithm 2 Reachability Analysis

```
INPUT: bi_set                ▷ a set of extracted Binder interfaces
OUTPUT: r_set                ▷ a set of reachable Binder interfaces
1: s_visited ← {}   ▷ a global set used to avoid infinite recursion in SEARCH
2: r_visited ← {}   ▷ a global set used to avoid infinite recursion in ANALYSIS
3: r_set ← ANALYSE(bi_set)
4: function ANALYSE(bi_set)
5:     tmp_set ← {}
6:     for all bi ∈ bi_set do
7:         if thenbi ∈ r_visited
8:             continue
9:         r_visited.add(bi)
10:        if IS_REGISTED(bi) then
11:            tmp_set.add(bi)
12:        if bi ∈ tmp_set then
13:            for all
                          func ∈ bi
 do
14:                for all param ∈ func do
15:                    if IS_OUTPARAM(param) then
16:                        s_visited ← {}
17:                        sub ← SEARCH(param)
18:                        tmp_set.union(subs)
19:                        tmp_set.union(ANALYSE(subs))
20:                    if IS_SRET(param) then   ▷ Check whether this param is a struct return of the function
21:                        s_visited ← {}
22:                        subs ← SEARCH(param)
23:                        tmp_set.union(subs)
24:                        tmp_set.union(ANALYSE(subs))
    return tmp_set
```

D REACHABILITY ANALYSIS ALGORITHM

E LIST OF VULNERABILITIES

Table 5: Vulnerabilities found and first reported by us. These vulnerabilities have different root causes and crash points. They are fixed in subsequent Android releases after our reports. Not all of them are assigned with CVE numbers as the Android security team prefers to fix as many bugs as possible in one patch.

CVE	CVSSv3 Score	Vulnerability Type	Affected TBI	Affected Function	Status
CVE-2020-0381	5.5	out-of-bound-write	IMediaExtractor	parse_wave()	Acknowledged (2020.9)
CVE-2020-0383	7.5	out-of-bound-write	IMediaExtractor	parse_ins()	Acknowledged (2020.9)
CVE-2020-0384	5.5	out-of-bound-write	IMediaExtractor	parse_art() and Convert_art()	Acknowledged (2020.9)
CVE-2020-0385	5.5	out-of-bound-write	IMediaExtractor	parse_rgn()	Acknowledged (2020.9)
CVE-2017-0665	7.8	out-of-bound-read	ISurfaceComposer	Surface::dequeueBuffer()	Acknowledged (2017.7)
CVE-2018-6241	7.8	arbitrary-write	ISurfaceComposer	NvGrRegisterBuffer()	Acknowledged (2019.1)
CVE-2019-9460	4.6	arbitrary-write	ISurfaceComposer	SoftRender::render()	Acknowledged (2020.6)
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	ARGBToRGB565Row_SSE2()	Fixed
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	I422ToRGB565Row_SSSE3()	Fixed
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	NvGrRegisterBuffer()	Fixed
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	SoftwareRenderer::render()	Fixed
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	gralloc_map()	Fixed
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	Surface::dequeueBuffer()	Fixed
N/A (dup)	N/A	arbitrary-write	ISurfaceComposer	ARGBToRGB565Row_SSE2()	Fixed
N/A (dup)	N/A	out-of-bound-write	ISurfaceComposer	Surface::attachBuffer()	Fixed
CVE-2017-0737	7.8	out-of-bound-write	IOMX	GraphicBufferSource::onFrameAvailable()	Acknowledged (2017.8)
N/A (rewarded)	N/A	info-leak	IOMX	GraphicBuffer::flatten()	Fixed
N/A (dup)	N/A	info-leak	IOMX	GraphicBuffer::flatten() (in gralloc.tegra.so)	Fixed
N/A (dup)	N/A	info-leak	IOMX	GraphicBuffer::flatten() (in gralloc.msm8996.so)	Fixed
N/A (dup)	N/A	info-leak	IOMX	GraphicBuffer::flatten() (in gralloc.default.so)	Fixed
N/A (dup)	N/A	use-after-free	IOMX	PersistentProxyListener::onBuffersReleased()	Fixed
N/A (dup)	N/A	use-after-free	IOMX	PersistentProxyListener::onFrameAvailable()	Fixed
N/A (dup)	N/A	use-after-free	IOMX	PersistentProxyListener::onSidebandStreamChanged()	Fixed
N/A (dup)	N/A	null-pointer-deference	IOMX	BpGraphicBufferConsumer::releaseBuffer()	Infeasible
CVE-2021-0426	7.8	out-of-bound-write	IStatsd	LogEvent::parsePrimaryFieldFirstUidAnnotation()	Acknowledged (2021.4)
CVE-2021-0427	7.8	out-of-bound-write	IStatsd	LogEvent::parseExclusiveStateAnnotation()	Acknowledged (2021.4)