NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage

Philipp Mao Marcel Busch Mathias Payer
EPFL, Lausanne, Switzerland

Abstract

Compromised or malicious apps remain a primary security concern for Android. As Android tightens its app sandbox and further reduces the kernel's attack surface, native Android system services emerge as a promising target for privilege escalation. Bugs in these native system services, triggerable from the app sandbox via RPC (Remote Procedure Calls), may facilitate privilege escalation.

We identify the attack surface exposed by proprietary native system services and propose NASS, an approach to effectively fuzz proprietary real-world RPC servers to detect bugs triggerable via RPC. NASS addresses the challenge of extracting coverage from complex intertwined real-world RPC servers. Furthermore, NASS leverages our novel technique *deserialization-guided interface extraction* to recover the RPC interface definition from proprietary RPC servers. NASS' techniques all build on common RPC design principles, which broadly apply to RPC frameworks.

We implement NASS for Android's Binder RPC framework. NASS outperforms prior work regarding interface extraction, target exploration and bug finding capabilities, even without access to source code. NASS has identified 12 unique bugs in up-to-date Google, Samsung, Xiaomi, and OnePlus devices, with five CVEs assigned so far.

1 Introduction

Smartphones have become the most intimate piece of technology. As such, smartphones are a promising target for malicious actors, as a compromised phone becomes a powerful spying tool. In the past, there have been multiple publicly documented instances where Android smartphones were compromised [17, 24]. A common exploitation pattern is an initial compromise of an internet-facing app, such as WhatsApp or Chrome, and then one or more privilege escalation steps often targeting a kernel bug to obtain root privileges. Root privileges enable spyware to stealthily harvest all the user's data unhindered by permission restrictions placed on apps.

To counteract the threat of compromised apps targeting the kernel, Google is sandboxing apps to limit direct access to the kernel. Apps only have access to a limited number of system calls and kernel drivers. In the past, attackers have still found ways to exploit the kernel despite the sandbox. However, directly attacking the kernel from an app is becoming increasingly difficult. As the Binder kernel driver, accessible by design from apps, is rewritten in Rust [33], attackers must look for other avenues to expose the kernel's attack surface.

Android system services are highly privileged user space daemons, which facilitate access to the kernel for apps. Apps interact with system services via Binder Remote Procedure Calls (RPC). Almost all Android framework APIs used by apps communicate with system services, which access the hardware on behalf of the app. While all apps are constrained by the same strict sandbox which restricts access to the kernel, each system service has its bespoke sandbox with access to specific kernel APIs. For performance reasons, many system services are native, and implemented in C++. On five recent devices, 528 (30% of all system services) are native system services. As RPC is used to allow communication from lower to higher-privilege contexts, vulnerabilities triggerable over RPC may be exploited to escalate an attacker's privileges. By exploiting a system service the attack surface exposed by the kernel is significantly increased as many services, by design, have access to more kernel APIs.

While prior work has identified the importance of securing native system services [8, 28], it has not effectively addressed *proprietary* system services. Out of the 528 native services mentioned above, 316 services are entirely proprietary and 44 are only partially open-source as they also load proprietary components. The security of these proprietary services is especially relevant since these are often part of the Hardware Abstraction Layer (HAL), with privileged access to the kernel. To address the security of proprietary native system services, we propose NASS, a fuzzer that can target **all** native Android system services. NASS addresses two challenges, interface awareness and coverage, not addressed by prior work.

The interface exposed by a system service defines the sig-

natures of functions exposed as RPCs. As pointed out by FANS [28], effectively fuzzing RPC servers is only possible with awareness of the server's exposed interface. Unfortunately, these interfaces are not publicly disclosed for proprietary RPC servers which poses a strict limitation for existing approaches [28]. NASS' primary contribution sets out to overcome this limitation.

NASS' approach is inspired by the generic design of RPC frameworks. Our major insight stems from three common RPC design principles (discussed in Section 3) that lead to the separation of automatically generated Inter-Process Communication (IPC) and (de)serialization logic on the one hand, and the manually-implemented application logic of the RPC on the other hand. Particularly the uniformity of auto-generated code, typically generated from domain-specific interface description languages, allows NASS to employ systematic dynamic analyses to *fully* recover the interface definitions from proprietary RPC servers. In detail, this technique called *deserialization-guided interface extraction* (*DGIE*) automatically recovers the complete interface definition from a proprietary RPC server by dynamically analyzing the server's execution of RPC framework-specific deserialization primitives when processing incoming requests. We implement DGIE for Android's RPC-over-Binder to recover the interface definition of proprietary system services without requiring source code or the necessity to capture IPC requests sent by the system. As a result, the knowledge of the target service's interface enables interface-aware fuzzing, allowing the fuzzer to generate inputs that are always successfully deserialized by the target RPC server. Additionally, it enables the creation of interface-aware mutators which mutate RPC arguments while adhering to their semantics.

Prior work has fuzzed system services in a black-box manner. System services are complex, multi-threaded processes, closely coupled with the rest of the system. Collecting stable coverage that is directly related to fuzzer dispatched RPCs is challenging. NASS enables grey-box coverage guided fuzzing of system services by leveraging the structure of RPC code to trace the service's execution only during the processing of fuzzing input using dynamic binary instrumentation.

We used NASS to fuzz proprietary system services on five recent commercial off-the-shelf (COTS) devices from Google, Xiaomi, Samsung, and OnePlus. So far, we have discovered 12 unique bugs with five CVEs assigned so far. We have responsibly disclosed all bugs to the affected vendors. NASS outperforms prior work in terms of interface recovery, coverage, and discovered bugs.

In summary, we present the following contributions:

- We are the first to identify the prevalence and importance of the attack surface exposed by proprietary native Android system services.
- Based on common RPC design principles, we present a novel technique, DGIE, leveraging dynamic analysis, to precisely extract the interface definition of proprietary APIs exposed over RPC.
- We propose an approach to extract stable coverage related to fuzzer dispatched RPCs from real-world multi-threaded RPC servers.
- We implement NASS, a fuzzer that leverages our insights, to efficiently fuzz any native Android system service. NASS is the first system that fuzzes proprietary system services with interface awareness and coverage.

We will open-source NASS' implementation to enable further research into Android system services and RPC fuzzing.

2 Android System Services

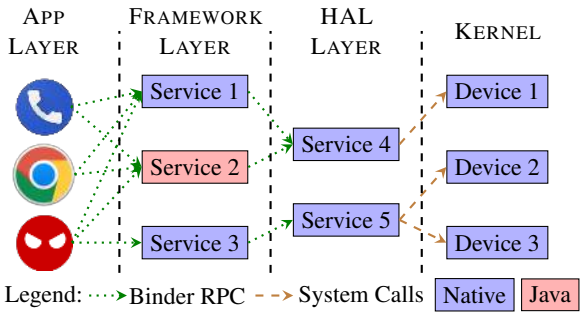

Figure 1: The user space architecture on modern Android devices with a binderized HAL.

The architecture of the modern Android user space consists of three layers: the app layer, the framework layer, and the hardware abstraction layer (HAL). Since Android 8, as part of project Treble, communication between the framework and HAL layer exclusively takes place over Binder IPC.

All user space processes running at any layer are isolated from all other processes with discretionary access control (Linux permissions), system call filtering (seccomp), and mandatory access control (SELinux).

Processes running at the app layer are user-facing applications (Android apps). Since apps may be malicious or contain vulnerabilities, they are subject to the most restricted sandbox. This sandbox restricts access to most kernel APIs, especially kernel drivers. To interface with the underlying hardware, apps communicate with system services, which in turn either communicate with other system services or the kernel. This interposition layer restricts hardware accesses to those implemented in the service and enables enforcement of Android permissions.

System services are user space processes running at the framework or the HAL layer. System services expose APIs over RPC that client processes can invoke. The Binder IPC kernel driver provides the RPC transport layer. System services running in the framework layer are the backend im-

plementations of the Android framework API used by apps. System services running in the HAL layer interact with the hardware directly (usually by communicating with a kernel driver) and are thus dependent on the underlying hardware. Note that a system service may belong to both the framework and the HAL layer.

To illustrate how system services in these different layers facilitate hardware access for an app, take for example an app that wishes to take a picture using the camera. The cameraserver service in the framework layer implements the backend for the standard `android.hardware.camera2` API. The app will first obtain a handle to the cameraserver service's `android.hardware.ICameraService` interface. Afterwards, the app uses the obtained handle to invoke the necessary RPC functions in the cameraserver system service. The cameraserver system service in turn communicates with the camera HAL system services over RPC. The camera HAL system services finally have the SELinux permissions to interact with the hardware-specific camera kernel driver and the camera hardware.

System services may be implemented in Java or natively in C++. System services may be open-source (part of the Android Open Source Project–AOSP) or proprietary. In general, all the system services running in the HAL layer are proprietary since these services are specific to the underlying hardware and are thus developed by the original equipment manufacturer or original design manufacturer. Vendors may extend the Android framework API (to support vendor-specific apps shipped on the device), resulting in proprietary system services running in the framework layer as well.

3 RPC Design Principles

RPC enables clients to execute API functions in other processes (RPC servers) as if they were local functions. An RPC server's interface definition contains the signatures of functions exposed over RPC. We identify three common RPC design principles, which apply to Android's RPC over Binder but can also be found in other RPC frameworks. In Section 6 we leverage these design principles to design DGIE, which automatically recovers the interface definition from proprietary RPC servers.

(Ab) Abstraction of IPC Binding Code It is common practice for developers to separate API-specific code and IPC-specific code. By abstracting away the underlying IPC mechanisms, developers can focus on the API functions without having to worry about IPC minutiae. It also makes the API functions portable between different IPC implementations and versions.

To facilitate this separation, IPC binding code is implemented in the client stub and the server stub. The client stub takes care of serializing the target function's arguments and sending the IPC request to the server process. When the client calls the remote API function, the client stub is invoked instead. The server stub is invoked after receiving an IPC request. It deserializes the function arguments and invokes the target API functions.

(Si) Single Entrypoint The entry point is the function invoked by the underlying IPC transport layer after receiving an IPC request. The entry point function maps the incoming IPC request to the target RPC interface and invokes that interface's server stub. Using a single entry point centralizes the handling of IPC requests, allowing the IPC transport layer to forward all incoming requests to the entry point without needing to parse its contents. This design separates the IPC transport layer from the IPC processing layer.

(St) Standard Deserialization Routines Deserializing input arguments from the incoming IPC request involves mapping input bytes back to higher-level types. These deserialization routines invoked in the server stub must be consistent with the client stub's serialization routines. Furthermore, the format of serialized arguments may depend on the underlying IPC mechanism. To ensure consistency and compatibility, both serialization and deserialization routines are standardized, either automatically generated or part of a runtime library available to both client and server.

In practice, RPC frameworks implement these design principles starting with an interface definition language (IDL). The developer defines the RPC interface in the IDL and the RPC framework's compiler takes care of emitting the client and server stub code. For instance, Android developers can use the Android Interface Definition Language (AIDL) to define the system service's interface. Two further propular RPC frameworks gRPC and Thrift, which we will discuss in Section 9, leverage their own IDLs.

4 Motivation

RPC calls (through Binder) cross a security boundary from one Android sandbox to another. These sandboxes have different privileges, giving access to other system services or kernel APIs. As such a vulnerability in a system service triggerable over RPC could be exploited to escalate privileges. In this section, we explain why system services are a promising target for attackers.

We assume an attacker who is executing code in Android's app layer and is looking to fully compromise the phone by elevating privileges to root. In Figure 1 the attacker is symbolized by the red icon. There are many ways our attacker could have gained code execution in the app layer. The attacker may have exploited a vulnerability in an app (e.g., Chrome 0-days, or one-click RCE exploits in WhatsApp [17, 24]) or the victim installed a malicious app. The attacker's compromised

app is running on a modern Android phone and is subject to correctly configured access control, seccomp and SELinux policies. In the past 10 years, attackers further elevated their privileges predominantly by exploiting a kernel vulnerability.

However, the attack surface of the kernel, which is directly accessible by a malicious app is shrinking rapidly. On modern phones, an app is sandboxed to a small subset of benign system calls and the Binder kernel driver. The Binder driver enables the IPC communication between apps and system services as well as system services between themselves. By design, apps need access to the Binder driver. This along with the complexity of the Binder driver has made it the prime target for privilege escalation in recent years [22, 23, 36, 42]. However, there is an active push to rewrite the Binder driver in Rust [33]. Once deployed, this change is expected to close this avenue for triggering bugs in the kernel. Furthermore, as more kernel mitigations, implemented in the mainline Linux kernel, trickle down to deployed phones [30], exploiting a kernel vulnerability becomes even more challenging.

One way for attackers to open up the attack surface of the kernel is to compromise a system service. System services are subject to the same restriction mechanisms as apps. However, by design, the SELinux policy applied to system services either grants them access to other system services or the kernel. Either type of service is an interesting target for our attacker. Compromising a HAL system service with extended access to the kernel (for example a vendor-specific kernel module) significantly increases the kernel's attack surface exposed to the attacker. Compromising a framework system service with access to HAL system services allows the attacker to directly communicate with these HAL system services and mount attacks against them. System services implemented in C++ are the most promising targets for such an attacker, since these system services may be vulnerable to memory corruption vulnerabilities, which can be exploited to achieve arbitrary code execution.

There is evidence that malicious actors have already started targeting Android system services to escalate privileges. In July 2024 a new entry was added to Google Project Zero's "0-day in the Wild" tracking sheet describing a vulnerability in a kernel driver on the Samsung S10 exploited on real phones [39]. What is notable about this vulnerability is that it was exploited not from an app but from a system service, namely the cameraserver service. The kernel driver containing the vulnerability was only accessible from the cameraserver process and the exploit was part of a privilege escalation chain. The cameraserver is a native system service that was likely exploited first and used as a stepping stone to exploit the kernel vulnerability.

While the cameraserver is part of the AOSP, many native system services are closed-source proprietary binary blobs. Inspecting the services running on five recent Android phones showed a large number of such proprietary services deployed. As Table 1 suggests, out of 528 native services, 316 (60%) are proprietary. Furthermore, out of the 212 native services that are part of the AOSP, 44 load proprietary binaries (21%). This includes the aforementioned cameraserver service. All this proprietary native code is exposed over RPC and is reachable either from an app or another service.

	Overall	Google Pixel 9	Samsung S23	Xiaomi RM Note 13	OnePlus 12R	Infinix X6710
# S.	1784	371	388	310	437	278
# N.S.	528	146	110	61	162	49
# P.N.S.	316	102	67	20	117	10
# F.P.N.S.	23	10	0	0	13	0
# H.P.N.S.	293	92	67	20	104	10

Table 1: An overview of the native (N) services (S) running on five recent COTS devices. 60% of all native services are proprietary (P), out of which 92% run in the HAL (H) layer and 8% in the framework (F) layer.

5 Challenges

Our goal is to find memory corruption bugs in proprietary native Android system services triggerable via RPCs. Fuzzing has proven to reliably find bugs in native code. However, applying fuzzing to complex, proprietary RPC servers presents two key challenges. As these challenges have so far not been solved, no effective fuzzing has been applied to native proprietary Android system services, leading to a blind spot in the security of the Android user space. This is especially critical since HAL system services, with the most access to the kernel, account for over 92% of proprietary native system services.

(C1) Automated Complete Interface Definition Extraction For Proprietary RPC Servers. The RPC interface definition defines the signatures of API functions exposed over RPC by the server. When an IPC request is received, the server stub attempts to deserialize the API function arguments from the IPC request's argument bytes (according to RPC design principle (Ab)). If the argument bytes cannot be deserialized to the expected arguments, the stub will return early without executing the target API function. In order to apply effective fuzzing to RPC servers, the fuzzer needs to be aware of the interface definition. This way the fuzzer can generate IPC argument bytes that are correctly deserialized by the server stub and will execute the target API functionality. Otherwise, the fuzzer will waste cycles fuzzing automatically generated stub code — which is likely correct. For open source targets the interface can be recovered from the IDL source file. However, *for proprietary RPC servers, the interface definition is unknown and has to be extracted.*

(C2) Isolating Coverage For Proprietary HW/SW-Dependent Multithreaded RPC Servers. Coverage-

guided grey box fuzzing has emerged as the predominant form of fuzzing [9, 41]. While the problem of extracting coverage from binary-only targets is mostly solved [10], extracting stable coverage that is directly connected to the sent IPC request from proprietary RPC servers is challenging. Android RPC servers usually have intricate hardware and software dependencies. For sufficiently complicated dependencies rehosting the RPC server in a dedicated fuzzing environment becomes prohibitively difficult (think back to our cameraserver example which needs access to the specific camera HAL services which in turn need access to the camera device driver) and thus the only way to dynamically execute the RPC server is in its original environment, with the expected hardware and software dependencies interacting with the server. Often RPC servers are multi-threaded programs that host multiple RPC interfaces. Naively collecting coverage from all threads for all received requests will lead to spurious coverage due to other system components interacting with the RPC server and instability due to the non-determinism of thread scheduling.

Prior Work on Android System Service Fuzzing. For Android's RPC over Binder two prior works propose solutions to fuzz native services, neither of which addresses (C1) or (C2). FANS [28] uses source code analysis to infer a service's interface definition. As this approach requires source code it does not apply to proprietary system services. BinderCracker [8] records IPC requests triggered during phone usage. It extracts partial information on the service's interface definition by inspecting the captured IPC request. While this technique applies to proprietary system services it is not exhaustive. It can only recover the expected arguments for RPC functions the system uses during normal operation, missing out on the attack surface exposed by rarely used or unused RPC functions. Both FANS and BinderCracker fuzz Android system services in a black box fashion without coverage feedback.

6 Design

We propose NASS, an approach that addresses both challenges to effectively fuzz real-world native RPC servers. NASS leverages the fact that RPC servers adhere to the three RPC design principles (abstraction of IPC binding code (Ab), single entry point (Si), and standard deserialization routines (St)) identified in Section 3.

NASS incorporates DGIE, which builds on RPC design principles (Ab) and (St), to extract the target server's interface definition addressing (C1). Furthermore, NASS enables evolutionary mutation-based grey box fuzzing of interdependent and multithreaded RPC servers by integrating stable coverage collection, which is collected from the single entry point (Si), addressing (C2). NASS' approach is applicable to all RPC frameworks compliant with these three design principles. In this work, we applied NASS to Android's RPC over Binder.

```
1  class SomeObj : public :: Parcelable{
2    std::vector<std::string> strings; int anInt; String aString;
3  }
4  status_t SomeObj::readFromParcel(){
5    if(!data.readStringVector(&strings)) return -1;
6    if(!data.readInt32(&anInt)) return -1;
7    if(!data.readString(&aString)) return -1;
8  }
9  status_t Demo::onTransact(
10   uint32_t code, const Parcel& data,
11   Parcel* rep, uint32_t flags) {
12   switch(code) {
13     case 1: {
14       SomeObj obj;
15       if(!data.readParcelable(&obj)) return -1;
16       return reply.writeInt32(f1(obj)); }
17     case 2: {
18       int32_t a; std::vector<int> b;
19       if(!data.readInt32(&a)) return -1;
20       if(!data.readInt32Vector(&b)) return -1;
21       return reply.writeInt32(f2(a,b)); }
22 }};
23 virtual int32_t f1(SomeObj obj) {
24   ...
25 }
26 virtual int32_t f2(int a, std::vector<int> b) {
27   ...
28 }
```

Listing 1: The source code of the example service. Brown lines indicate automatically generated server-stub code while teal lines show application-specific, manually written logic. The service exposes two RPC functions, `f1` and `f2`. The service's entry point function is `Demo::onTransact`. Figure 2 shows the service's interface defintion (including the definition of `SomeObj`) and how NASS extracts it.

For both extracting the interface definition and coverage collection, NASS uses dynamic analysis. The dynamic analysis only uses addresses of standard library symbols or addresses previously extracted by NASS' dynamic analysis, making NASS' design applicable to binary-only targets.

To motivate our design decisions, we will use a running example of a toy native Android system service. Our example system service is shown in Listing 1 and exposes two functions over RPC.

6.1 Coverage Collection for RPC Servers

NASS leverages coverage for both extracting the target RPC interface and fuzzing. RPC servers are often multi-threaded programs with at least one thread handling incoming IPC requests for a given interface. A single server process may host more than one RPC interface. Other system components may be invoking RPCs on the other interfaces hosted in the target server's process. Both interface extraction and fuzzing rely on coverage information that is stable and directly related

to the sent IPC request.

Conforming with RPC design principle (Si), every RPC server's interface has an associated entry point function, which is invoked by the IPC transport layer after receiving an IPC request destined for that interface. To extract coverage of sent IPC requests, NASS hooks the entry point function. Every time a client sends an IPC request to the target interface, NASS' hook is triggered. By inspecting the IPC request NASS can deduce if the IPC request was sent by itself or an unrelated system component. In the latter case NASS' hook returns and lets the server continue normally. Otherwise, the hook starts tracing the current thread, updating the coverage bit map for each encountered basic block, until the entry point function returns. This way NASS can guarantee that coverage is only collected for code relevant to handling its own IPC requests.

For our example service in Listing 1, the entry point function is `Demo::onTransact`. Coverage for this service will be collected when this function is called until it returns.

6.2 Deserialization-Guided Interface Extraction

Starting from the entry point function, the server attempts to deserialize the input arguments based on the target RPC function's signature. After successfully deserializing the expected arguments, the target RPC function is called. The invocation of this server stub after receiving an IPC request follows RPC design principle (Ab). The server stub enforces the interface definition by ensuring that the received argument bytes can be deserialized to the expected arguments. For proprietary RPC servers, the interface definition is not known in advance. By dynamically analyzing the server stub, NASS automatically infers the interface definition for proprietary RPC servers leveraging DGIE.

In the first step, NASS identifies all exposed RPC functions. The target RPC function is typically identified either by a numeric value (as in Android Binder IPC, which uses an integer whose most significant byte is masked) or by a string (as used in gRPC or Thrift) in the IPC request. In both scenarios, the set of possible identifiers is finite. In the case of a string identifier, the string is stored in the target server's binary. Iterating over all strings in the server binary will eventually find the RPC identifiers. NASS iterates through these identifiers, sending IPC requests to the target server and monitoring coverage. Whenever new coverage is observed, it indicates the discovery of a new RPC function. NASS then records the corresponding identifier for future use. For the example service in Listing 1 NASS identifies two RPC functions (identifiers `1` and `2`) after iterating over all possible 3-byte values.

Following RPC design principle (St), the server stub utilizes standard routines to deserialize RPC arguments. NASS identifies the address of these routines and observes their execution. In the case of Android system services, these routines

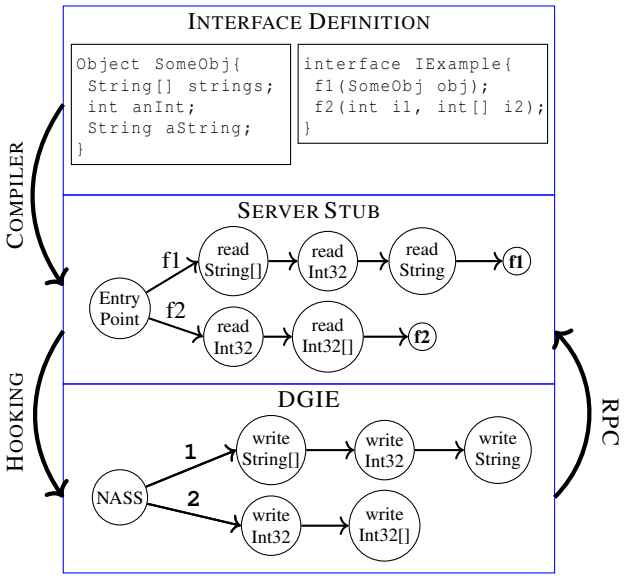

Figure 2: An overview of how NASS uses DGIE to extract the interface definition from the compiled server stub by hooking it and invoking RPCs. The target interface contains two functions `f1` and `f2`. The argument of the `f1` function is a high-level object, which unrolls to a linear sequence of deserializers in the server stub.

are exported by the `libbinder.so` and `libbinder_ndk.so` libraries, whose serialization and deserialization routines are defined in the AOSP in `Parcel.h` and `binder_parcel.h` respectively.

RPC function arguments are deserialized one after another, observed as either a single invocation of a deserialization routine or a sequence of multiple deserialization routines (for object argument types). In both cases, NASS can observe the invoked deserializers and use the standard serialization routines to generate RPC argument bytes that are successfully deserialized by the observed sequence of deserializers. NASS iteratively invokes RPC functions, using the previously discovered identifiers, observes the deserialization routines and updates the extracted interface definition until no more new deserialization routines are observed, implying that all RPC function arguments were deserialized successfully and the target RPC function has been called. NASS associates the observed sequences of deserializers with the used RPC function identifier. The sequence of used deserializers maps to a deserializer-level RPC function signature. While NASS cannot recover the high-level objects for RPC function arguments, in the server stub these objects are unrolled to a linear sequence of standard deserializers. To recover the deserializer-level function signature for `f1` for the example service from Listing 1, NASS first invokes `f1` over RPC with random bytes as the argument and observes the invocation of the `readString[]` deserializer. NASS adds this deserializer to its interface for identifier `1` and then in-

vokes `f1` again with argument bytes consisting of a serialized `String[]` and random bytes. This time NASS observes first a call to `readString[]` and `readInt`. This is repeated until NASS has extracted all three deserializers needed to call `f1`. Figure 2 gives an overview of this process.

As long as the server stub follows RPC principles (Ab) and (St) the sequence of deserializers for any RPC function is unrolled to a linear sequence of standard deserializers, which NASS is guaranteed to recover. NASS repeats the same enumeration for all previously discovered RPC functions until it has fully extracted the interface definition. The extracted interface definition maps each RPC function to a sequence of serialization functions that mirror the sequence of deserialization functions used in the server stub.

6.3 Interface Aware Fuzzing

With a way to collect coverage and having extracted the interface definition, NASS has all the pieces in place to efficiently fuzz the target RPC interface. NASS leverages the extracted interface definition to generate seeds that adhere to the server's interface definition. NASS' fuzzing component implements an interface-aware mutator, which only mutates the value of the arguments of the RPC call without changing the structure of the request. By adhering to the extracted interface definition, NASS can effectively target the functions exposed over RPC with the expected types without wasting cycles on inputs that get rejected by the server stub.

The mutator is aware of the argument types and implements mutations that adhere to the semantics of the type. Given RPC design principle (St), the standardized deserialization functions used by the server also have a counterpart serialization function. NASS uses these to serialize the mutated argument values in the format expected by the server stub. Mutated and serialized RPC arguments are sent to the target server over the supported IPC transport channel and coverage is collected starting from the entry point function until its return. The coverage is fed back to NASS and used for seed scheduling.

7 Implementation

NASS' implementation is designed to target any Android native service deployed on COTS devices. Since system services are often closely coupled to the underlying hardware, we fuzz services in-situ in their original environment. This way any crashes found by NASS' implementation are true positive crashes and not an artifact of the rehosting environment. After crashing a service we rely on Android's init process to automatically restart and correctly initialize the target service. NASS' implementation requires root permissions to run. Since system services are user space processes, the root user has full control over these processes. NASS' implementation runs as root and uses dynamic binary instrumentation (DBI) to hook and trace the target service. We build our DBI on top of FЯIDA [32].

NASS is implemented in three modules. The instrumentation module is injected into the target service process. Its job is to hook and inspect the service's state at runtime and during fuzzing to collect coverage. The client component is responsible for sending IPC requests to the target service and when fuzzing mutating IPC requests, reading the coverage bitmap, and scheduling seeds. Both the aforementioned modules run on the device. The orchestrator module runs on the host and sets up both the instrumentation and client modules over the Android Debug Bridge (ADB).

To apply NASS to a target service, NASS only requires the Binder handle which serves as an identifier to send IPC requests to the service. NASS queries the `ServiceManager` service, to whom all Android processes have a handle and which is responsible for distributing service handles, to obtain the handle to the target system service.

7.1 Entrypoint Identification

To hook the entry point function its address has to be known. The entry point function's address of a given RPC interface can generally not be directly inferred from the service binaries' symbols. In Android system services the entry point function is called the `onTransact` function. NASS' instrumentation module automatically extracts the address of the target service's `onTransact` function by hooking Android's user space Binder library at the point when execution is handed over to the `onTransact` function. The hook is triggered by NASS' client module sending an IPC request to the service.

7.2 Coverage Collection

NASS' instrumentation module hooks the `onTransact` function. Every time a client sends an IPC request to the target interface the hook is triggered. To collect coverage only for IPC requests sent by NASS' client module itself, the hook inspects the IPC request and checks if the process id (PID) of the caller is the same as the client module's PID. If so, the instrumentation module starts tracing the current thread's execution using FЯIDA Stalker until the `onTransact` function returns. Every time a new basic block is encountered, the coverage bitmap shared between the instrumentation and client module is updated. With this technique, we achieve between 30 to 400 executions per second when fuzzing on COTS devices.

7.3 Interface Extraction

NASS' implementation iteratively applies DGIE to extract the service's interface definition. DGIE itself is split into two phases: a fuzzing phase to extract a preliminary interface definition and a refinement phase to extract its deserializers. For

Ⓐb-compliant server stubs, this implementation is functionally equivalent to DGIE as described in Section 6.2, providing the same guarantees, while allowing to probabilistically explore server stubs violating Ⓐb. We found instances of developers violating RPC design principle Ⓐb, intertwining application-specific logic with the server stub. This potentially introduces control-flow dependencies in the server stub based on deserialized values. By first fuzzing the server stub, mutating the structure of the serialized argument bytes and their values, NASS can trigger code that has control-flow dependencies on the deserialized values.

Preliminary Interface Definition To extract a preliminary interface definition, NASS fuzzes the RPC server. Coverage is collected as described in Section 6.1. IPC requests that triggered new coverage are saved to the seed corpus and mutated further. In the context of NASS, a seed stores one IPC request. These seeds contain the RPC function identifier and the serialized argument bytes. Seeds are mutated by changing the function identifier or by adding/removing/mutating a serialized argument. New coverage implies one of three scenarios: the server stub was able to deserialize more of the expected arguments, new application-specific code in the server stub was triggered, or the new coverage is related to the functionality of the RPC functions themselves.

The resulting seed corpus contains the preliminary interface definition, which maps RPC functions to byte arrays of serialized arguments. Note that this byte array maps to many possible sequences of deserializers, which may not match the deserializers used in the server stub. In the example system service from Listing 1, an `Int64` RPC argument with value 0 may be correctly deserialized to call `f2`. Note that the first four bytes map to the `Int32` and the next four bytes encode the length of a 0-sized array. However, fuzzing `f2` with a `Int64` argument will lead to many rejected inputs due to the strict enforcement of the length value encoding of the integer array. This one-to-many relationship of serializing functions (whose output bytes are successfully deserialized by one or multiple deserializing functions) allows the fuzzer to easily generate IPC requests covering the deserializers in the server stub.

Refining the Interface Definition To identify the exact deserializers, NASS refines the initial interface definition by monitoring which deserialization routines are triggered when replaying the IPC request seeds generated during fuzzing. Each time the server invokes the deserialization routines, NASS refines the preliminary interface definition by associating a subset of the IPC request's argument bytes with the deserializing function used. After replaying all IPC requests, NASS has observed all covered deserializers along with the order of invocation. Replaying the `Int64` with value zero, discovered during preliminary interface fuzzing of the example service, against `f2` NASS observes an invocation of `readInt32` and `readInt32[]`.

Iterative DGIE To extract the complete RPC interface definition, NASS has to cover the entire server stub during preliminary interface fuzzing, which allows triggering all deserializers during refinement. To estimate the point at which the server stub has been covered, NASS iteratively extracts a preliminary interface definition, refines it, and then uses it to seed the next preliminary interface fuzzing phase until, for each RPC function identifier, no new deserializers are observed during refinement. During refinement NASS also replays the seeds from previous iterations to avoid any loss of information. In the first iteration, NASS fuzzes the RPC function identifier, iterating over all possible values to identify the exposed RPC function identifiers. In subsequent iterations, NASS only mutates the argument bytes to avoid redundancy.

To terminate the preliminary interface fuzzing phase, NASS observes the rate at which new coverage is observed and proceeds to the next step when plateauing. In its current implementation NASS uses the number of new seeds in the first two minutes and compares this to the number in subsequent two-minute intervals, moving to refinement if the new number of seeds is ten times lower than the initially observed rate. Additionally, the current implementation unconditionally moves to refinement after 20 minutes of fuzzing. While refinement is done on a per-RPC function identifier basis, for fuzzing, we rely on the fuzzer's scheduler, which can target all RPC function identifiers due to being seeded in every round with at least one seed per exposed RPC function.

For the example service, NASS may extract the partial interface definition after the first iteration for `f1` (`readString[]` and `readInt32`) but fails to cover `readString`. In the second round, due to seeding the fuzzer with the partial interface definition, NASS discovers an input that is successfully deserialized to a `SomeObj` object, covering all three deserialization routines required to call `f1`. Another iteration of fuzzing and refinement will not trigger new deserializers, indicating that the complete interface was extracted.

7.4 Interface Aware Mutations

There are 23 different argument types supported by NASS for Binder IPC. Each type is serialized with the corresponding standard serializer function and mutated based on the type. Table 2 provides an overview of all types and mutators.

To mutate primitive types such as integers and strings, NASS uses standard mutators that operate on bytes. Vectors of primitive types are mutated by either mutating individual vector entries or by removing or adding vector entries.

There are two special types for which the fuzzing input is not sent in the IPC request itself, file descriptors, and binder references (StrongBinder). A file descriptor argument is mutated by writing the fuzzing input to a temporary file and

Type	Mutators Used
Bool, Char, Short, Int32, Int64, Float, Double	FixedLengthBytes, InsertSpecialNumber
String8, String16, CString	FixedLengthBytes, VarLengthBytes, InsertSpecialString
BoolVector, CharVector, ShortVector, Int32Vector, Int64Vector, FloatVector, DoubleVector	ChangeVectorSize, MutateEntryFixed
String8Vector, String16Vector	ChangeVectorSize, MutateEntryVarLength
StrongBinder	MutateStrongBinder
FileDescriptor	MutateFileDescriptor
Parcelable	All*
ParcelableVector	ChangeVectorSize, All*

Table 2: The types supported in Android's RPC over Binder and the corresponding mutators used by NASS.

serializing the file descriptor. The target service may then read or write to the file descriptor sent over IPC. A Binder reference is a handle to another system service most often used to implement a callback, notifying the client of the status of a previously initiated task. Such callbacks usually only send data back to the client. NASS' mutator handles the rare case that a Binder reference is used to transfer more data from the client to the service. The mutator serializes a reference to a bespoke fuzzing input service. The fuzzing input service returns mutated fuzzing input when receiving an IPC request.

Parcelables are high-level objects that can be serialized and sent over Binder IPC. No general Parcelable deserialization function exists, instead deserializing a single Parcelable object is done with a fixed sequence of standard deserializers. Thus NASS transparently handles Parcelables as sequences of standard deserializers. Mutating vectors of Parcelables is done either by mutating any one of the Parcelables members or by adding/removing Parcelable vector entries. Adding or removing entries to a Parcelable vector is done by inserting or removing the sequence of argument types corresponding to a single Parcelable entry.

7.5 Fuzzing The Service

To fuzz the service, either during DGIE or after having extracted the interface definition, the orchestrator module first injects the instrumentation module into the target system service and uploads the client module to the device under test. Afterward, the client module is started in fuzzing mode. The client is based on LibFuzzer [29] and implements custom RPC and interface-aware mutators. Mutated IPC requests are sent to the target service via the Binder kernel driver. Crashes are detected by monitoring the target service's PID after sending an IPC request. If the service crashes the orchestrator module takes care of reinstrumenting the restarted service and restarting the client module. It also periodically restarts the service to reset the service's state.

8 Evaluation

Evaluating NASS, we answer these research questions:

- **RQ1:** Can DGIE *fully* recover RPC interfaces from binary-only RPC server stubs? — Our first novelty claim.
- **RQ2:** Does NASS' interface awareness improve its coverage and bug-finding capability? — Our second novelty claim.
- **RQ3:** Does NASS' approach improve the state-of-the-art in Android system server fuzzing in terms of exploration, bug-finding, and interface recovery capabilities? — Evaluating the effectiveness of our system.
- **RQ4:** Can NASS find bugs in proprietary system services deployed on modern COTS devices? — Demonstrating the usefulness of NASS on COTS devices.
- **RQ5:** Do proprietary system services adhere to the three RPC design principles (Ab), (Si), and (St)? — Justifying the design of DGIE.

8.1 Experimental Setup

Our experimental setup aims to demonstrate NASS' bug-finding and code exploration capabilities, which are established metrics in fuzzer evaluations. Additionally, we leverage a manually created ground truth of open-source system service interfaces to evaluate the accuracy of DGIE.

We compare NASS to FANS [28], a fuzzer that targets open-source native Android system services. FANS uses static source code analysis to infer the interface definition for open-source native system services. It then fuzzes the analyzed services using a black box fuzzer. To collect coverage from FANS for comparison with NASS, we extend FANS' fuzzer component to collect coverage for the generated seeds using dynamic binary instrumentation, similar to NASS. We leverage FANS' original seed generation algorithm to ensure a fair comparison. Thus even though there is coverage, FANS generates its seeds as it did originally, in a black box fashion. With NASS' ability to recover coverage from system services, we are able to measure the coverage achieved by FANS' approach for the first time, five years after its publication.

Another pertinent work is BinderCracker [8]. Unfortunately, BinderCracker's prototype is not available. While FANS' evaluation against BinderCracker simply compared the absolute number of detected bugs, we evaluate BinderCracker's approach of message capturing against NASS' self-driven approach of DGIE on our ground-truth dataset.

We conduct these experiments on 14 open-source native system services supported by FANS and NASS running on Android 9 (the Android version originally used in FANS' evaluation). We conduct this evaluation on Android 9 to allow for an unbiased and fair comparison against FANS. In

Service	AIDL	# RPC functions	# NASS disc. RPC funcs.		# NASS extr. RPC funcs.		# Capt. RPC funcs.	
keystore	no	42	42	100%	36	85%	27	64%
gatekeeper	no	6	6	100%	3	50%	4	66%
perfprofd	no	5	5	100%	4	100%	0	0%
stats	no	18	17	94%	17	94%	8	44%
wificond	yes	14	14	100%	14	100%	9	57%
surfaceflinger	no	50	42	84%	32	64%	23	46%
netd	yes	37	37	100%	37	100%	17	45%
installd	yes	39	39	100%	39	100%	27	69%
vold	yes	51	51	100%	51	100%	25	49%
gpu	no	1	1	100%	0	0%	0	0%
media.metrics	no	2	1	50%	1	50%	1	50%
storaged	yes	3	3	100%	3	100%	3	100%
thermalservice	yes	4	4	100%	4	100%	1	25%
incident	no	4	3	75%	1	25%	3	75%
Overall	6/14	276	265	96%	242	88%	148	53%

Table 3: Results of the ground truth study of DGIE. Extracted RPC functions is the number of RPC functions for which NASS extracted the correct sequence of deserializers. Captured RPC functions are RPC functions which were observed to be used by the system while running the Android CTS.

their original paper, the authors of FANS discuss the difficulty of porting FANS to other Android versions. Porting FANS to the newest Android version would inevitably degrade FANS' quality leading to skewed results. Furthermore, between Android 9 and the newest Android version, the user space components of Android's RPC framework over Binder only underwent minimal changes, and NASS works out of the box for Android 9. We stress that NASS was designed to run on the newest Android version and for our real-world bug-finding evaluation we run NASS on devices with Android versions from 13 to 15 (currently the most recent version).

The experiments to evaluate the contribution of NASS' techniques and to compare NASS to prior work are conducted on the Android emulator [31] running Android 9.0.0_r46 for arm64 with KVM support. The Android emulator accurately runs the Android user space, executing the system services at native speed with KVM, and allows for quick resets required for long-running evaluation experiments. The arm64 host on which we run the emulator uses an NXP Layerscape LX2160A CPU with 32GB RAM and 16 cores. The experiments evaluating NASS' ability to target real-world proprietary system services are conducted on five recent COTS phones. These five phones are from five different market-leading vendors (Google Pixel 9, Samsung S23, Xiaomi Redmi Note13, OnePlus 12R, Transsion Infinix X670), fully updated running an Android version between 13 and 15, and use a variety of SoCs (system on chip).

8.2 Interface Ground Truth Study

To understand the accuracy achieved by DGIE, we manually analyze the server stubs of all 14 evaluation services and compare the manually analyzed stubs to NASS' extracted interface definitions. To compare against BinderCracker's message capturing approach, we log RPC functions triggered during phone usage. In this way, we estimate how much of the actual interface BinderCracker can recover. To exercise the phone, we use the Compatibility Testing Suite (CTS) [14], which vendors use to ensure that devices are compatible with the Android framework APIs. We ran the full test suite, which took 60 hours, and in total captured 3 million RPC function invocations. See Table 3 for the results.

NASS extracts 265 out of 276 exposed RPC functions and recovers the correct sequence of deserializers for 242(88%) of them. Six out of 14 services use automatically generated server stubs. For these services NASS recovers the correct sequence of deserializers for 100% of RPC functions.

NASS' interface extraction struggles for services where developers include application-specific logic in the server stub such as the `SurfaceFlinger`, `gatekeeper`, `incident`, `gpu` and `media.metrics` services, only recovering the correct deserializer sequence for 58% of RPC functions. These services violate RPC design principle (Ab). Thus NASS is not guaranteed to recover the complete interface definition. However, as the coverage evaluation in Section 8.3 will show, even FANS' source code-based approach struggles with these services. For the remaining three non-AIDL services NASS extracts the correct sequence of deserializers for 90% of RPC functions. Even though these services do not use autogenerated server stubs, developers still mostly adhered to RPC design principle (Ab).

Relying on dynamically capturing IPC messages only recovers 53% of RPC functions, compared to NASS' 88%. Thus BinderCracker's approach vastly underestimates the exposed attack surface.

> **RQ1.** DGIE recovers complete interface definitions for system services adhering to RPC principles (Ab).
> **RQ3.** Relying on capturing IPC requests vastly underestimates the exposed attack surface, only recovering 53% of all RPC functions compared to DGIE's 88%.

8.3 Coverage And Bug Finding

We now measure NASS' exploration and bug-finding capabilities. These experiments include an ablation study including a non-interface-aware configuration of NASS, NASS (NI), and the comparison against the state-of-the-art system server fuzzer FANS. NASS (NI) uses standard byte-level mutators without knowledge about the IPC argument types. This baseline allows us to quantify the effects of NASS' interface awareness.

We conduct fuzzing experiments using the 14 evaluation services from Section 8.1 to compare NASS, NASS (NI), and FANS. We fuzz each service five times for 12 hours. To measure coverage, we replay the seed queue against the target service and use a modified version of NASS' coverage instru-

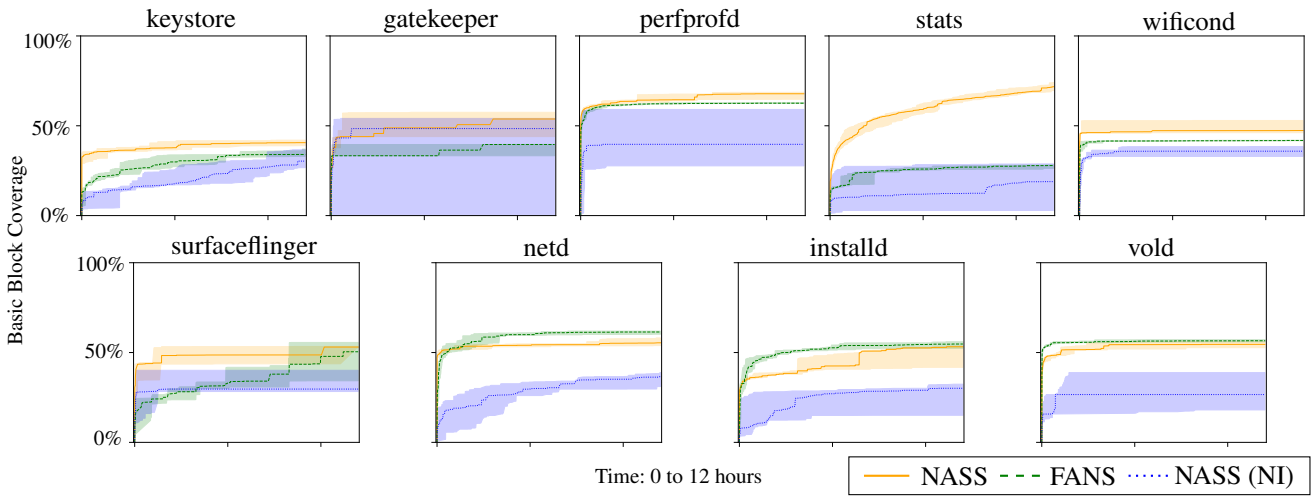

Figure 3: Basic-block coverage observed over 12 hours of fuzzing across nine of the 14 evaluation services. The plots for the remaining five services can be found in the Appendix Figure 4. NASS (NI) is the non-interface-aware configuration of NASS. The shaded regions indicate min and max coverage achieved by the respective fuzzer across five runs.

Service	# Crashes / # Unique Crashes					
	NASS		FANS		NASS(NI)	
wificond	0	0	0	0	0	0
netd	0	0	2	2	0	0
installd	4	0	11	7	2	0
storaged	0	0	0	0	0	0
thermalservice	0	0	0	0	0	0
gatekeeper	2	0	2	0	2	0
incident	1	0	2	0	2	0
media.metrics	0	0	0	0	0	0
gpu	0	0	0	0	0	0
surfaceflinger	0	0	3	3	0	0
perfprofd	1	1	0	0	0	0
stats	2	0	4	2	2	0
keystore	1	1	0	0	0	0
vold	8	7	1	0	0	0
Sum	19	9	25	14	8	0

Table 4: The manually deduplicated crashes detected by NASS, FANS and, NASS without interface-awareness (NI). Unique crashes were only detected by one fuzzer.

mentation to collect the addresses of executed basic blocks. Additionally, we determine a conservative lower bound for the maximally achievable coverage per service by leveraging control-flow reachability. Concretely, we analyze each service (i.e., main binary and libraries) using the Ghidra decompilation framework and extract the control-flow graphs for all functions in which we covered at least one basic block. Then, we establish the maximally achievable coverage by counting the total number of basic blocks *over all* function control-flow graphs. Figure 3 shows the coverage graphs of the fuzzing campaign where 100% indicates the conservative lower bound of achievable coverage, and Table 4 lists the deduplicated crashes discovered by each fuzzer across five runs for the fuzzed services. See Appendix Table 8 for the absolute number of basic blocks for each service.

Five evaluation services have an interface with less than four RPC functions and the RPC functions themselves have very few basic blocks. As NASS and FANS start out by already covering most of the services' exposed code, the coverage graphs show both NASS and FANS almost instantly plateauing. The coverage graphs for these services can be found in the Appendix Figure 4. For these services NASS' non-interface aware configuration is able to compensate for missing interface awareness with coverage-feedback, even outperforming FANS on four services. NASS' interface extraction failed to identify one of the two RPC functions for the `media.metrics` service, see Table 3, leading to NASS' low coverage.

Overall NASS outperforms FANS in terms of coverage for eight out of the 14 services and achieves parity for four out of the remaining six services. While FANS leverages sophisticated source code analysis and manual variable-to-semantics mapping, NASS achieves similar results or even outperforms FANS using automatic binary-only analysis, relying on coverage and hooking. Furthermore, NASS outperforms its non-interface-aware configuration for all except two services (`gatekeeper` and `media.metrics`). These two services violate RPC design principle Ⓐb causing NASS' extracted interface definition to be incorrect and hampering NASS' ability to explore the service.

In terms of bugs discovered during the campaign, NASS discovered 19 and FANS discovered 25, while NASS without interface awareness only discovered 8. Out of the 19 bugs

discovered by NASS, 9 were only discovered by NASS, and for two services NASS was the only fuzzer to discover bugs. Out of the bugs listed in FANS' paper, we were able to identify 13 affecting our evaluation services. From these 13 original FANS bugs, NASS discovered 11 of them. The two bugs missed are in the `surfaceflinger` service, for which NASS' interface extraction struggled due to the service violating RPC design principle Ⓐb.

NASS struggles to achieve the same coverage and/or number of bugs discovered as FANS for services that impose very strict semantic requirements on input parameters (`netd` and `installd`), such as file paths or IP addresses, returning early or aborting the process if these semantic requirements are not met. FANS satisfies these strict requirements by applying a hardcoded set of heuristics dependent on the RPC argument name. For instance, an RPC argument `file` is populated with a path pointing to an existing file on the system. Since NASS' primary focus is closed-source system services where the original name of RPC arguments is removed in the compilation process, it cannot rely on such heuristics. Hence, NASS usually violates strict semantic requirements, which hampers its exploration and bug finding ability for a small set of services. However, adding FANS-style mutation heuristics to NASS is only limited by the amount of reverse-engineering effort an analyst is willing to spend on accurately identifying the semantics of RPC arguments. This effort, while important for fuzzing in production, is beyond the scope of this research.

RQ2. NASS outperforms its configuration without interface-awareness in terms of coverage and bug-finding capability.
RQ3. Compared to FANS' source-code based approach NASS achieves parity in terms of coverage and discovers bugs missed by FANS.

8.4 Real-World Bug Finding

We fuzzed 316 proprietary services on five devices to evaluate NASS' ability to find bugs on COTS devices. Table 6 shows the results of this fuzzing campaign.

In our campaigns, NASS discovered 2590 crashes. As NASS fuzzes in a persistent manner we build up state during fuzzing, which can make reproducing crashes difficult. We attempt to reproduce the crashes by replaying the crashing seed and if the crash is not reproduced we replay the seed corpus generated before the crashing seed. This way we can reproduce 2029 crashes. We deduplicate the crashes based on the backtrace and manual analysis resulting in 72 unique, reproducible crashes. We consider all of these crashes bugs as they can be triggered over the IPC boundary from another process' sandbox. We differentiate between bugs that simply cause a service to crash, such as null pointer dereferences or C++ exceptions, and bugs that may potentially be weaponized to achieve a useful memory corruption primitive. NASS discovered 60 and 12 such bugs respectively. Out of the 34 services where NASS found non-exploitable crashing bugs, 26 are compliant with RPC design principles. One of the 11 services where NASS found a memory corruption bug is not compliant with RPC design principles. Table 5 gives an overview of the discovered memory corruption bugs. These bugs could be weaponized to, e.g., achieve arbitrary code execution in affected services. We have responsibly disclosed the discovered bugs to the vendors. So far, five CVEs have been assigned. We present three case studies of the discovered bugs to demonstrate their severity.

Use After Free, Pixel 9. This vulnerability affects the `android.hardware.radio.sap.ISap/slot2` service on the Pixel 9. The RPC function exposed over command ID 1 allows clients to dispatch an apdu request. Each apdu request is stored in a linked list, identified by a token provided as an RPC argument. After completing the request the service traverses the linked list and frees the first apdu request matching the token. When handling command ID 1, the vulnerable service does not check for duplicate tokens. NASS triggered the crash by sending two requests to command ID 1 with the same token, causing the service to free an apdu request still in use after completing the first request, leading to a use after free. This use after free could have been exploited to achieve code execution in the `rild_exynos` process running as the privileged radio user. It was assigned CVE-2024-47040 by Google and fixed in November 2024.

Arbitrary Write, OnePlus 12R. This vulnerability affects the `vendor.oplus.hardware.engineer.IEngineer` service on the OnePlus 12R. The vulnerable RPC function sets an integer value in a memory buffer using the user-provided index and value without any range check, leading to an arbitrary memory write (in integer range).

Heap Overflow, Samsung S23. This vulnerability affects the `vendor.samsung.hardware.radio.network` service on the Samsung S23. One of the RPC functions deserializes a network protocol message from the IPC request (using a sequence of seven different deserializers). A field of this message is the message length, a signed integer. The deserialized message is later copied to an internal buffer for dispatching using the attacker-controlled length. Even though the service makes sure the value is not larger than the maximum message size, the check is signed. As a consequence, a negative length value will lead to a heap-buffer-overflow.

RQ4. NASS' interface-aware and coverage-guided fuzzing discovers vulnerabilities on real-world COTS devices.

Device	Service	Bug Type	Disclosure Status	Assigned Severity	CVE
Pixel 9	vendor.google.battery_mitigation.IBatteryMitigation	OOB read	fixed	high	CVE-2025-0085
Pixel 9	android.hardware.secure_element.ISecureElement	OOB read	fixed	medium	CVE-2024-56186
Pixel 9	android.hardware.radio.sap.ISap	UAF	fixed	high	CVE-2024-47040
Pixel 9	android.hardware.boot.IBootControl	OOB read	fixed	high	CVE-2024-47039
Pixel 9	android.hardware.radio.config.IRadioConfig	stack overflow	fixed	high	CVE-2025-26459
Pixel 9	android.hardware.radio.sim.IRadioSim	OOB read	disclosed	low	N/A
Pixel 9	android.hardware.radio.sim.IRadioSim	heap overflow	fixed	high	pending
Samsung S23	vendor.samsung.hardware.radio.network.ISehRadioNetwork	heap overflow	disclosed	high	pending
Redmi Note 13	miui.whetstone.klo†	invalid unmap	disclosed	none	N/A
OnePlus 12R	vendor.oplus.hardware.fido.fidoca.IFidoDaemon	OOB read	disclosed	N/A*	N/A
OnePlus 12R	vendor.oplus.hardware.engineer.IEngineer	OOB write	disclosed	none	N/A
OnePlus 12R	vendor.oplus.hardware.urcc.IUrcc	OOB write	disclosed	pending	N/A

Table 5: The memory corruption vulnerabilities discovered by NASS. (*)Fixed independently of us. (†)Not compliant with RPC design principles (St).

Device	Android Version	# Crashes	# Crashes Repr. & Ded.	# DoS Bugs	# M.Corr. Bugs
Pixel 9	15	934	28	21	7
Samsung S23	14	568	11	10	1
Redmi Note 13	13	312	12	11	1
OnePlus 12R	15	736	15	12	3
Infinix X670	13	40	6	6	0
Overall		2590	72	60	12

Table 6: The results of the fuzzing campaign on five up-to-date COTS devices.

Device	# Services	# (Si)	# (Ab)	# (St)	# Compliant	
Pixel 9	102	102	95	95	95	93%
Samsung S23	67	67	62	63	61	91%
Redmi Note 13	20	20	16	13	12	60%
OnePlus 12R	117	117	106	109	104	88%
Infinix X670	10	10	9	9	9	90%
Overall	316	*316	288	289	281	89%

Table 7: The results of the RPC design principle compliance analysis. (*) Android's IPC over Binder enforces (Si).

8.5 Compliance of COTS Services with RPC Design Principles

To understand if RPC design principles apply to COTS services, we manually analyzed the server stub of all 316 proprietary native services. Due to Android's Binder IPC mechanism, all services are compliant with RPC design principle (Si). For compliance with (Ab), we check that there are only control-flow dependencies due to standard deserialization routines, not application-specific logic. For compliance with (St), we recursively follow the serialized argument byte array object and ensure that it is only processed by standard deserialization routines. We conservatively mark a service as non-compliant if any principle is violated. Table 7 shows the results of the analysis. 281 services (89%) are compliant with all three RPC design principles. We found that 28 manually written server stubs violated (Ab) by including permission checks, logging, or application-specific checks on deserialized arguments.

The (St) principle was broken by 27 server stubs with custom deserialization instantiating objects outside of the standard Parcelable routines. The level of compliance is comparable between vendors except for Xiaomi (60% compliance).

RQ5. Proprietary services deployed on COTS devices largely adhere to all three RPC design principles.

9 Discussion

Extending DGIE to Other RPC Frameworks. While NASS' current implementation of DGIE is specialized to work for Android system services, DGIE is applicable to other RPC frameworks that comply with the RPC design principles as outlined in Section 3. In Appendix A.1 we discuss how two established RPC frameworks gRPC [13] and Thrift [1] adhere to these design principles. Both gRPC and Thrift are widely established in industry, used among others by Netflix, OpenAI, Facebook, and Uber [2,11]. While DGIE applies to these RPC frameworks there are implementation details that need to be taken into account. These details include identifying of entry point addresses and deserialization routines, and mapping of instrumented deserializers back to the RPC argument.

Coverage Collection for Other Threads. NASS collects coverage only for the thread handling incoming IPC requests and only while the incoming IPC request is processed. This way NASS obtains stable and service-related coverage from arbitrary system services. This approach limits the coverage collection for threads that asynchronously handle fuzzer input. The handling thread may put part of the fuzzing input into a queue, which is then processed by another thread. Identifying such cross-thread dependencies in binary-only programs and handling coverage collection for asynchronous input processing is an unsolved challenge that we leave to future work.

Nested Interfaces. Android system services might not directly expose their entire RPC interface. Instead, these nested interfaces are retrieved over RPC from the top level interface.

NASS' implementation currently does not support automatically identifying and obtaining handles to nested interfaces. However, with a handle to a nested interface, NASS can extract its interface definition and fuzz it. We leave automatically identifying and obtaining handles to nested interfaces in proprietary Android system services as future work.

Limitations of On-Device Fuzzing. NASS relies on DBI to introspect proprietary system services running on-device. During fuzzing, DBI incurs a 30x overhead compared to sending IPC requests in a loop without instrumentation. State-of-the-art fuzzers [9, 10] reduce the overhead from DBI by caching rewritten basic blocks. With additional engineering effort, this approach could be implemented for NASS' coverage instrumentation. An orthogonal approach is to rehost COTS services in an emulated environment, enabling horizontal scaling without physical devices. However, rehosting proprietary Android components is an open research problem, orthogonal to NASS' contributions.

During fuzzing, state accumulates. This state may persist a restart of the service or even a system reboot. The state buildup can result in non-reproducible crashes, which makes triaging much more challenging. It is however important to note that all crashes triggered by NASS are true positives as they were triggered via the target's exposed interface.

NASS does not use a sanitizer. There are binary-only implementations of ASAN [10] to detect heap memory corruptions. NASS's instrumentation module attaches to already running services, which makes applying ASAN retrospectively challenging since the heap state would need to be recovered.

Finally, NASS runs on rooted phones, limiting the COTS devices that can be targeted without having to first manually find a privilege escalation vulnerability.

10 Related Work

As system services are an important component of Android's architecture, a number of works have studied various security aspects of system services. Orthogonal to our work, Elgharabawy et. al, examined the use and security of Unix domain sockets as the IPC transport medium instead of Binder [7]. One focus of prior work are DoS attacks capable of effectively shutting down the system, triggered through system services [18, 37, 44]. While NASS' goal is finding exploitable memory corruption bugs, the DoS bugs found by NASS may be leveraged to effectively turn off the target service and thus deny other user space components access to specific features. System services are the enforcing mechanism of Android permissions and many works have investigated confused deputy attacks on system services. Works such as [15, 16, 19, 35, 45] analyze system services to find paths to privileged functionality without correct permission checks. Xiang et. al. [38] identify the attack surface exposed by Binder references, wherein services temporarily act as clients, sending IPC requests back to the original clients. NASS takes such bugs into account with its bespoke Binder reference mutator, which returns fuzzing input when responding to IPC requests sent to previously serialized Binder references.

Closely related to our efforts, there have been a number of works on fuzzing native Android system services. First to fuzz this attack surface and show that exploiting vulnerabilities in native system services over Binder IPC to escalate privileges is possible was Gong [12] in 2015. BinderCracker [8] records IPC requests sent by the system to extract partial interface definitions to fuzz either the Java or native services. While BinderCracker's insights into unrolling Parcelables using hooking are similar to ours, we map this insight to widely used RPC principles and also demonstrate that a hooking-based approach can extract complete interface definitions. Most recent, FANS [28] fuzzes open source native system services leveraging source code analysis to extract the target service's interface. Unlike NASS none of the above approaches incorporate coverage for fuzzing. Furthermore, NASS is the only system capable of extracting the complete interface definition from binary-only native system services. While there have been many works investigating the security of Android system services, until now none have addressed the attack surface exposed by proprietary native system services.

More broadly researchers have studied the automatic inference of expected argument types to improve fuzzing efficiency. Static analysis has been applied to a variety of targets including user space libraries [20, 21], kernel drivers [4–6], and secure monitors [27]. Another approach is dynamically capturing requests sent to the target [3, 40, 43] to infer the interface definition. Machfuzzer [40] targets proprietary RPC servers, however, it does not leverage our insights into common RPC design principles and relies on message capturing.

Compartmentalized software relies on RPC to facilitate communication between compartments [26]. One approach to discover compartment-interface vulnerabilities, triggerable over RPC from one compartment to the other, is fuzzing. Existing fuzzers targeting compartment-interface vulnerabilities such as NYX-NET [34] or ConfFuzz [25] could be extended with DGIE to enable interface-aware fuzzing without relying on source code.

11 Conclusion

NASS introduces an approach to fuzzing proprietary real-world RPC servers with coverage and interface awareness. We implemented NASS for Android RPC over Binder to fuzz proprietary system services, a critical attack surface for privilege escalation on Android that so far had been neglected. Our evaluation uncovered 12 bugs on up-to-date COTS devices, with five assigned CVEs. We open source NASS to enable further research on RPC fuzzing and Android system services.

12 Ethics Considerations

As part of our evaluation, we discovered vulnerabilities on COTS phones. These vulnerabilities could be weaponized to target real people. We understand the criticality of our findings, and we have disclosed all vulnerabilities discovered by NASS immediately after discovery to the responsible vendors. The five assigned CVEs serve as proof of our dedication to responsible disclosure. At the time of publication, all affected vendors had at least 90 days to address the disclosed vulnerabilities.

We present novel techniques to help discover vulnerabilities. We are aware that these techniques could also be leveraged by malicious actors. We contend that security through obscurity is not a sustainable strategy for safeguarding software. By openly sharing our findings, we empower organizations and researchers to harden native RPC servers, reducing the overall attack surface available to malicious actors.

13 Open Science

We open source NASS and NASS' evaluation setup. We also participate in the artifact evaluation with the goal of obtaining all three badges. NASS is available at https://doi.org/10.5281/zenodo.15577630 or https://github.com/HexHive/NASS.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868) and SNSF PCEGP2 18697.

References

[1] Apache. Apache Thrift. https://thrift.apache.org/, 2007.

[2] Apache. Powered By Apache Thrift. https://thrift.apache.org/about#powered-by-apache-thrift, 2007.

[3] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1204–1219, 2023.

[4] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 749–763, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693, 2021.

[6] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Mounir Elgharabawy, Blas Kojusner, Mohammad Mannan, Kevin R. B. Butler, Byron Williams, and Amr Youssef. SAUSAGE: Security Analysis of Unix domain Socket usAGE in Android. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 572–586, 2022.

[8] Huan Feng and Kang G. Shin. Understanding and Defending the Binder Attack Surface in Android. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 398–409, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[10] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, CCS '22. ACM, November 2022.

[11] Cloud Native Computing Foundation. gRPC, Customer Success Stories. https://www.cncf.io/case-studies/?_sft_lf-project=grpc, 2015.

[12] Guang Gong. Fuzzing Android System Services by Binder Call to Escalate Privilege. In *Black Hat USA*, August 2024. Conference talk.

[13] Google. gRPC. https://grpc.io, 2015.

[14] Google. The Compatibility Test Suite (CTS). https://source.android.com/docs/security/bulletin/pixel/2024-11-01, 2025.

[15] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 25–36, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Sigmund Albert Gorski and William Enck. ARF: identifying re-delegation vulnerabilities in Android system services. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 151–161, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Google Threat Analysis Group. Buying Spying, Insights into Commercial Surveillance Vendors. https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Buying_Spying_-_Insights_into_Commercial_Surveillance_Vendors_-_TAG_report.pdf, 2024.

[18] Heqing Huang, Sencun Zhu, Kai Chen, and Peng 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*, CCS '15, page 1236–1247, New York, NY, USA, 2015. Association for Computing Machinery.

[19] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. FReD: Identifying File Re-Delegation in Android System Services. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1525–1542, Boston, MA, August 2022. USENIX Association.

[20] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, August 2020.

[21] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. UTopia: Automatic Generation of Fuzz Driver using Unit Tests. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2676–2692, 2023.

[22] Moshe Kol. BadSpin, CVE-2022-2042. https://github.com/0xkol/badspin?tab=readme-ov-file.

[23] BFS Labs. Part 2: Escalating to Root, CVE-2020-0041. https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/, 2020.

[24] Clement Lecigne. Spyware Vendors Use 0-days and N-days against Popular Platforms. https://blog.google/threat-analysis-group/spyware-vendors-use-0-days-and-n-days-against-popular-platforms/, 2023.

[25] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software . In *2023 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2023.

[26] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. SoK: Software Compartmentalization . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2884–2903, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[27] Christian Lindenmeier, Mathias Payer, and Marcel Busch. EL3XIR: Fuzzing COTS Secure Monitors. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5395–5412, Philadelphia, PA, August 2024. USENIX Association.

[28] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 307–323. USENIX Association, August 2020.

[29] LLVM. libFuzzer. https://llvm.org/docs/LibFuzzer.html, 2016.

[30] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4517–4534, Philadelphia, PA, August 2024. USENIX Association.

[31] Android Open Source Project. Android Emulator Source. https://android.googlesource.com/platform/external/qemu/+/refs/heads/master, 2025.

[32] Ole André V. Ravnås. FЯIDA. https://frida.re, 2013.

[33] Alice Rhyl and Carlos Llamas. Using Rust in the Binder Driver. In *Linux Plumbers Conference 2023*, Richmond, VA, USA, November 2023. Conference talk.

[34] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 166–180, 2022.

[35] Yuru Shao, Qi Alfred Chen, Z. Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *Network and Distributed System Security Symposium*, 2016.

[36] Android Red Team. How to Fuzz Your Way to Android Universal Root: Attacking Android Binder, CVE-2023-20938. https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938/offensivecon_24_binder.pdf, 2024.

[37] Kai Wang, Yuqing Zhang, and Peng 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*, CCS '16, page 92–103, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Xiaobo Xiang, Ren Zhang, Hanxiang Wen, Xiaorui Gong, and Baoxu Liu. 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, page 1581–1597, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Clement Lecigene Xingyu Jin. CVE-2024-44068: Samsung m2m1shot_scaler0 Device Driver Page Use-after-free in Android. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2024/CVE-2024-44068.html, 2024.

[40] Kun Yang, Hanqing Zhao, Chao Zhang, Jianwei Zhuge, and Haixin Duan. Fuzzing IPC with Knowledge Inference. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 11–1109, 2019.

[41] Michal Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2014.

[42] Google Project Zero. Bad Binder: Android In-The-Wild Exploit, CVE-2019-2215. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html, 2019.

[43] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2811–2828. USENIX Association, August 2021.

[44] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. Exploit the Last Straw That Breaks Android Systems. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2230–2247, 2022.

[45] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1165–1178, New York, NY, USA, 2018. Association for Computing Machinery.

A Appendix

A.1 RPC Frameworks

To illustrate how RPC design principles (Ab), (Si) and (St) apply to other RPC frameworks discuss how they apply to gRPC and Apache Thrift, two established RCP frameworks.

A.1.1 GRPC

GRPC [13] uses the proto IDL to define the RPC interface. From the IDL file the gRPC compiler generates source and header files which can be included by the client and server. In the server code the gRPC server is started and the RPC interface is registered with the server. (Si): The gRPC server's `SyncRequest::Run` function serves as the single entry point processing the input from the transport layer (HTTP/2 in the case of gRPC). (Ab): The server stub is comprised of gRPC server code for identifying the target RPC interface/method and then the libprotobuf code to parse the received protobuf message. The programmer does not touch any of this code, only registering the implementations of the RPC functions with the gRPC server. (St): Protobuf messages are described with a `TcParseTable`, which is traversed in a loop in the `google::protobuf::internal::MergeFromImpl` function. In each iteration, the next relevant deserializer function in libprotobuf is called to parse the next field. See Listing 2 for an example gRPC server along with the relevant code snippets of the server stub.

A.1.2 Apache Thrift

Thrift [1] uses its own IDL to define the RPC interface. From the IDL, the Thrift compiler generates source files that contain the server stub. The server stub is contained in the `Processor` class which is registered with the `ThriftServer`. (Si): The `dispatchCall` function of the `Processor` class is the entry point function invoked by the server when receiving an IPC request to the interface. (Ab): The server stub starting from `dispatchCall` to the invocation of the target RPC function is fully autogenerated. (St): Each RPC function has a dedicated

```
service Demo { rpc Test (TestReq) returns (Testrep) {} }
message TestReq { string reqmsg = 1; }
message TestRep { string repmsg = 1; }
```

```
class DemoImpl : Demo::Service {
  Status Test(ServerContext* context, const TestReq* req,
    const TestRep* rep) override {
      rep->set_repmsg("Hello!");
  }
}
int main(){
  ServerBuilder builder;
  builder.RegisterService(&service);
  Server server(builder.BuildAndStart());
  server->Wait();
}
```

```
void grpc::Server::SyncRequest::Run(...) {
  ...
  auto* handler = method_->handler();
  // Deserialize the Request
  deserialized_request_ = handler->Deserialize(
                              call_, request_payload_,
                              &request_status_, nullptr);
  if (!request_status_.ok()) {
    VLOG(2) << "Failed to deserialize message.";
  }
  ...
  ContinueRunAfterInterception(); //calls DemoImpl
}
```

```
bool google::protobuf::internal::MergeFromImpl(
  absl::string_view input, MessageLite* msg,
  const internal::TcParseTableBase* tc_table,
  MessageLite::ParseFlags parse_flags
) {
  ...
  while(1){
    tc_entry = (void *)TcParseTableBase::entry(tc_table);
    deserializer = (code *)TcParseTableBase::target(tc_entry);
    // calls into libprotobuf
    result = (char *)(*deserializer)(msg);}
  ...
}
```

Listing 2: An example gRPC server. The topmost box contains the .proto file, written by the developer which defines the RPC interface and protobuf objects. The second box is the server main code which implements the actual API functionality and starts the gRPC server. The third box lists the entry point function and the server stub. Based on the target RPC interface, the protobuf message is deserialized. If this succeeds, the `ContinueRunAfterInterception` is called which eventually calls the `DemoImpl` function. The final box contains the deserialization loop, which iterates over the `TcParseTable` table extracts the needed deserializer and calls the deserializer function in libprotobuf.

`read` function to deserialize its arguments. This function uses the deserializers exported by the `TProtocol` object, which

```
service Greeter { i32 greet(1:string message) }
```

```
Class GreeterHandler : virtual public GreeterIf {
  int32_t greet(std::string message){
    printf("geetings");
  }
}
int main(){
  ...
  GreeterHandler handler = handler(new GreeterHandler())
  Tprocessor processor(new GreeterProcessor(handler));
  TSimpleServer server(processor, ...);
  server.server();
}
```

```
bool GreeterProcessor::dispatchCall(Tprotocol* iprot,
          Tprotocol* oprot, std::string fname, int seqId, ...){
  // find the target RPC function handler
  pfn = processMap_.find(fname);
  pfn->second(seqId, iprot, oprot, ..); // call process_*
  ...
}
bool GreeterProcessor::process_greet(int seqid,
                    TProtocol* iprot, Tprotocol* oprot, ..) {
  Greeter_greet_args args;
  args.read(iprot);
  try{
    iface_->greet(args.message); // call RPC function
  ...
}
```

```
int Greeter_greet_args::read(TProtocol* iprot){
  int fid;
  while(true){
    iprot->readFieldBegin(fid);
    switch (fid) {
      case 1:
        iprot->readString(this->message);
    }
  }
}
```

Listing 3: An example Thrift service. The topmost box is the Thrift IDL file. The second box contains the server code, which starts the server along with the class containing the developer-written implementation of the RPC functions. The third box shows the relevant parts of the server stub. The `dispatchCall` function is the entry point for the RPC interface. On line 4 it maps the target RPC function to the handler, `process_greet` in the case of the `greet` RPC function. On line 11 of the `process_greet` function the arguments are deserialized and on line 12 the target RPC function is called. The final box is the deserialization function for deserializing the arguments to the `greet` function. Each argument is deserialized with standard deserializers from the `TProcotol` class.

holds the RPC argument bytes. Listing 3 shows the relevant code for an example Thrift service.

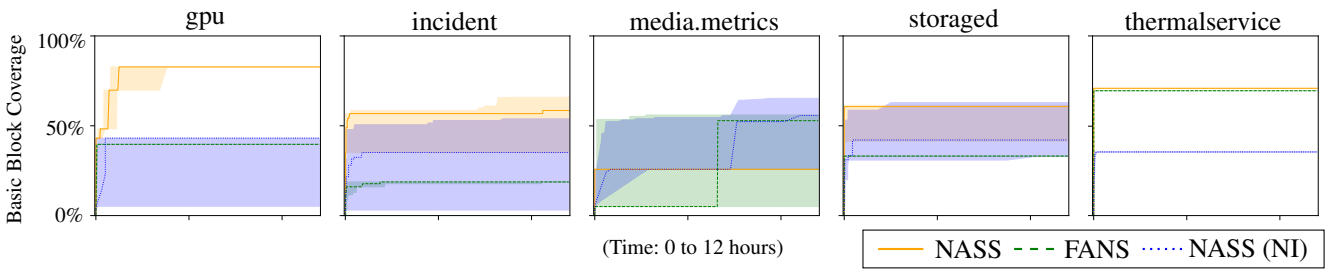

Figure 4: Basic block coverage observed over 12 hours of fuzzing across five of the 14 evaluation services. These five services all have an interface with less than four exposed RPC functions and the RPC functions themselves have very few basic blocks.

A.2 Coverage Graphs

Figure 4 shows the coverage graph for the five services not shown in Section 8.3. Table 8 shows the maximum number of reachable basic blocks and the basic blocks covered by each fuzzer.

Service	# Reach. BBs	# NASS Max. Cov. BBs	# FANS Max. Cov. BBs	# NASS (NI) Max. Cov. BBs
keystore	10589	3536	3087	3118
gatekeeper	2113	1010	707	955
perfprofd	2009	1159	1061	1001
stats	9847	6173	2334	2392
wificond	2208	988	779	717
surfaceflinger	6918	3394	3527	2549
netd	4472	2047	2140	1363
installd	6456	2715	2783	1676
vold	6070	2643	2677	1829
gpu	116	78	38	41
media.metrics	757	153	353	410
storaged	770	405	256	415
thermalservice	223	128	126	60
incident	518	290	82	238

Table 8: The number of reachable basic blocks and maximally achieved number of basic blocks.