STAB Fuzzing: A **St**udy of **A**ndroid's **B**inder IPC and Linux/Android **Fuzzing**

Travis W. Peters
Dartmouth College
CS258: Advanced Operating Systems

March 16, 2016

Abstract

This paper focuses on describing the necessary background to begin working with Binder: Android's Interprocess Communication (IPC) mechanism, and Linux/Android system call ("syscall") fuzzing tools. The objective was to study Android and Binder along with system call fuzzing in order to learn more about Android, Binder IPC, and vulnerability detection and analysis. Our study was further concentrated on the `ioctl()` syscall due to the significant role is plays in handling Binder data and the potential for abuse/misuse.

This paper will present our findings in studying the Android OS, specifically the Binder framework, as well a review of existing fuzzing frameworks, as well as describe our efforts in running Trinity-based fuzzing tools against Linux and Android.

1 Introduction

When I initially worked to define the scope of my project, my goal was to learn about Android OS internals and vulnerability detection and analysis. In my preliminary efforts to learn about Android OS it became apparent that Binder, Android's predominant IPC mechanism, is really the backbone of its architecture. Thus, I set out to better understand the details of Android's Binder implementation and to gain a better understanding of how IPC works in Android in general.

It was also suggested to me that, since I wanted to learn more about vulnerability detection and analysis in a practical sense, I should explore fuzzing tools. While fuzzing and fuzzing tools would be of interest in genera, it was advised that a particularly interesting subset of fuzzers to learn about would be *system call fuzzers*; more specifically, system call fuzzers for Android OS. Fuzzing, generally speaking, is the act of sending (semi-)random, or otherwise invalid, data as input to some entity (e.g., system, application, protocol parser). Fuzzing has long been a technique carried out by many hackers and responsible testers seeking to find vulnerabilities in how a particular component receives and handles (i.e., parses or deciphers) input it is given. Fuzzing, as I have found, is a vital form of testing that should be carried out in the development cycle of any/all applications, protocols, operating systems, and so forth; basically, anything that accepts and attempts to parse/interpret data should undergo strenuous fuzz testing to ensure "safe" behavior even under "unlikely" inputs.

In the following paper I present my work in (1) studying the implementation and code related to Android's Binder interface [3, 9], (2) present a large survey of fuzzing with historical information about the progression of fuzzing technologies as well as address many of the work/projects that have been done in the fuzz testing domain with special attention given to system call fuzzing and Android OS/application-related fuzzing, and (3) describe my efforts to run the Trinity fuzzing tool [12] on Linux to understand some of the workings of the tool. Complications with the development environment and a not-actively maintained or well-documented code-base have made it infeasible to successfully run a fuzzer against the Binder interface in the timeframe allotted for this project.

All of this work has been done to attain prerequisite knowledge that I believe is necessary in order to effectively use existing software, or write new software, that will enable stress-testing of the Binder IPC mechanism on Android, which will hopefully aid in identifying errors in the parsing of buffers that are passed around by the `ioctl()` system call ("syscall") in the underlying Binder kernel driver. This is left to future work.

Acknowledgement: *I thank Sergey Bratus for his help in developing the direction of this project and for all of his insights and tips that he shared via email at all hours throughout my time working on this project.*

2 Binder (IPC for Android)

The Android operating system (OS) consists of a fairly complex and layered software stack (see: Figure 1). At its core, Android OS utilizes the Linux kernel (and in some cases extends it or removes from it). Above the Linux kernel, Android OS implements various libraries and the Android Runtime; atop these is the Application Framework, which most application developers are familiar with since applications (apps) themselves run atop the Application Framework.

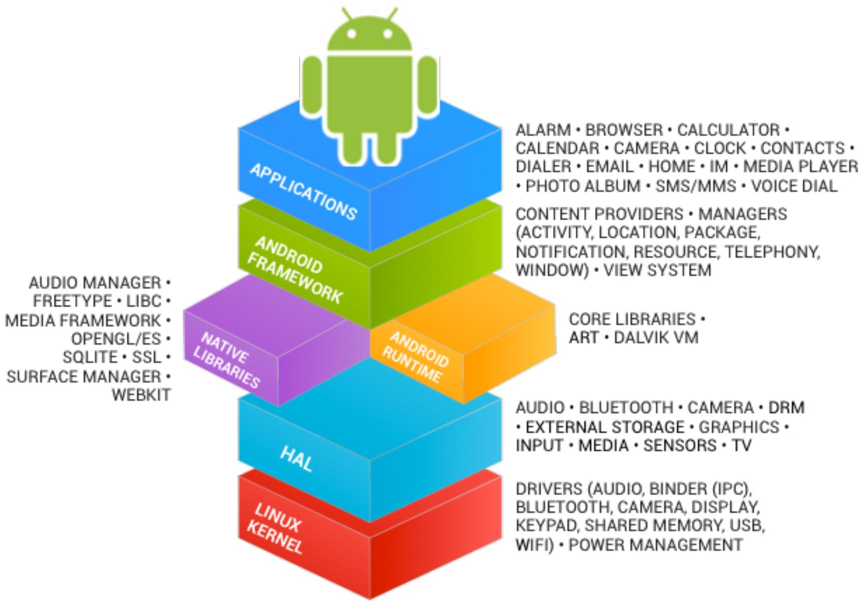

APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

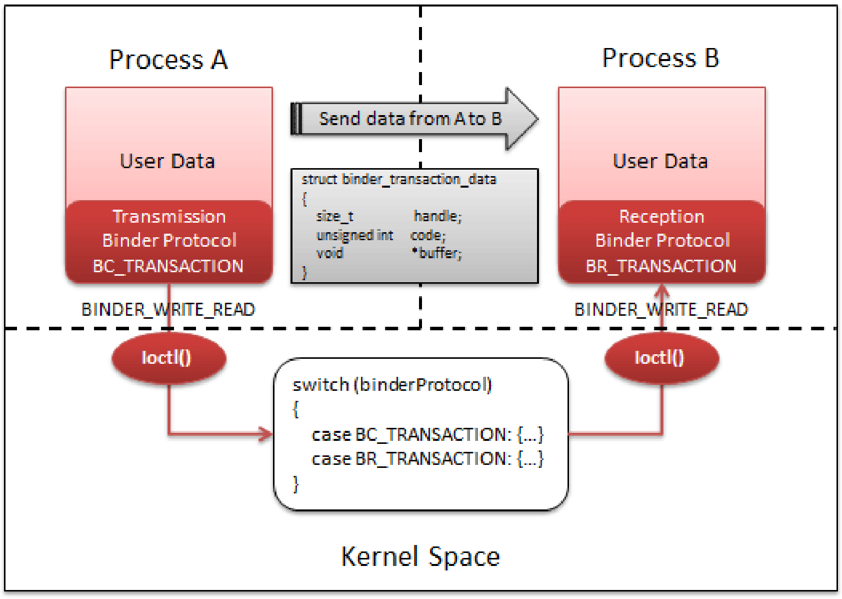
Figure 1: An overview of the Android software stack [21] that makes up what is commonly referred to as the Android operating system (OS).

Arguably one of the most important components in Android is *Binder*: Android's Interprocess Communication (IPC) mechanism. Unlike, more traditional systems (e.g., desktops and laptops) that run Linux and use typical Linux IPC mechanisms such as signals, pipes, sockets, shared memory, semaphores, and so forth to communicate with other processes, Binder is a fairly new[1] IPC mechanism that grew out of OpenBinder [24] in the earlier 2000s and has since been adopted by Google (the OpenBinder project, as far as I can tell, is no longer active though it lives on in the Android OS). According to the OpenBinder documentation, Binder is *"... a system-level component architecture, designed to provide a richer high-level abstraction on top of traditional modern operating system services,"* which, in the end, boils down to a nice IPC mechanism that

[1]The kernel-side component of the Linux version of OpenBinder was merged into the Linux kernel mainline in kernel version 3.19, which was released on February 8, 2015 [13].

enables communication with a remote service as if it existed in the same process space.

The Binder IPC mechanism effectively implements a client/server model. A client initiates communication by sending a Binder command (`BC`) and waits (i.e., is paused while waiting) for a response from a server. The server eventually responds with a Binder response (`BR`). Each communication between two processes via Binder is known as a *transaction*. This model is achieved through a 3-tiered design which include (1) an API for applications, (2) middleware that enables interaction with the kernel, and (3) the actual Binder kernel driver[2]. An overview of how data is actually transferred from one process to another via Binder is shown in Figure 2 and is discussed in greater detail below.



```
struct binder_transaction_data
{
  size_t          handle;
  unsigned int    code;
  void            *buffer;
}
```

```
switch (binderProtocol)
{
   case BC_TRANSACTION: {...}
   case BR_TRANSACTION: {...}
}
```

Figure 2: An overview of how data is sent from one process (A) to another process (B) via the Binder interface that ultimately dispatches to a call to `ioctl()` in the Binder kernel driver [4].

At the highest level (tier) we have the *Android Interface Definition Language (AIDL)* and the Java/Java Native Interface (JNI) wrappers. AIDL is a language that makes implementing remote services in Android easier. An AIDL file must be shared between remote service app developers and client app developers because the necessary Java code is generated from that file and enables communication. The Java/JNI layer serves to wrap the middleware and kernel driver.

[2]The real "magic" of Binder's implementation is found in the kernel driver source code: http://lxr.free-electrons.com/source/include/uapi/linux/android/binder.h and http://lxr.free-electrons.com/source/drivers/android/binder.c.

The middleware layer of Binder, written entirely in C++, provides access to the necessary routines and data structures which can do the heavy-lifting of Binder IPC. Specifically, the Binder middleware handles important tasks such as:

- the creation and management of worker threads,
- the serialization and reconstruction of complex objects to/from simple data types (e.g., `Int`, `Float`, `Boolean`, `String` types)—also known as marshalling and unmarshalling of Parcels, and
- direct interaction with the kernel driver via `ioctl()`.

At the heart of Binder IPC is the kernel driver. The binder kernel driver, written in C, supports file operations such as `open`, `mmap`, `release`, `poll`, and, of interest to us, `ioctl`. As expected, `open` and `release` handle opening/closing a connection with the driver itself, and `mmap` maps Binder memory, however most of the "action" happens through the `ioctl` system call; the `ioctl` syscall takes as arguments a Binder command (in the form of an integer code) and a reference to a data buffer. In response to a valid `ioctl` call, the kernel will fill in various meta-information about the call (e.g., PID of caller) and invoke an appropriate thread to copy data to/from the appropriate process space(s).

The Binder communication model and related terms/concepts are described further in [2, 22].

3 Fuzzy Fuzzing Fuzzer

With the exception of "fuzzy," the words in the title of this section ("fuzzing" and "fuzzer") are generally meaningful terms in the context of system and application testing. "Fuzzing" typical refers to the action of running a "fuzz test" whereas a "fuzzer" generally denotes a tool that is used to carry out the fuzzing of a particular entity. In this section we will review the basic idea behind what fuzzing really is as well as various fuzzers (i.e., tools used for fuzzing).

The general idea behind "fuzzing" is to generate (semi-)random data and send it to some interface as input. The interface could belong to an application, a library, the kernel (i.e., the system call—or syscall—interface), a network stack/protocol, and so forth. In the case of sending random data, there is no real effort to control the inputs in any way. These sorts of naïve fuzzers were helpful in identifying errors in input validation/handling in the past, however, many interface/protocol developers today are wise to the sorts of trivial bugs that occur when failing to do at least some basic input validation. This leads us to the idea of semi-random data fuzzing that consists of data that is "good enough" to look like valid data (as far as basic input validation is concerned), but is "bad enough" so that it could still identify errors in how the input data is handled and used.

NOTE: Technically speaking there are two classifications of fuzzing: *mutational* and *generational*. Mutational fuzzing can be thought of as taking known, valid, input and applying different types of *mutations*, then using the mutated input as a test case against some target. Generational fuzzing on the other hand is the process of *generating* inputs randomly to use as test cases against some target, and usually doing so in light of specific formats, interfaces, known (basic) validation testing, and so forth (i.e., not really "random", but rather, "semi-random").

One semi-random fuzzer that is of particular interest to us is the *Trinity* fuzzer [12]. Trinity is a syscall fuzzer that makes a great deal of effort to respect the basic specifications of the syscall interface (e.g., semi-random inputs use valid data types, valid file descriptors, valid memory addresses) in order to pass basic input validation and, as a result, enable better fuzz testing of the more error-prone data handling/parsing/manipulating that is done later on in most syscalls. More information about Trinity will by covered in the next section and in section 4.

A typical flow diagram depicting how fuzz testing should be carried out is depicted in Figure 3. The rest of this section is dedicated to looking at various types of fuzzers.

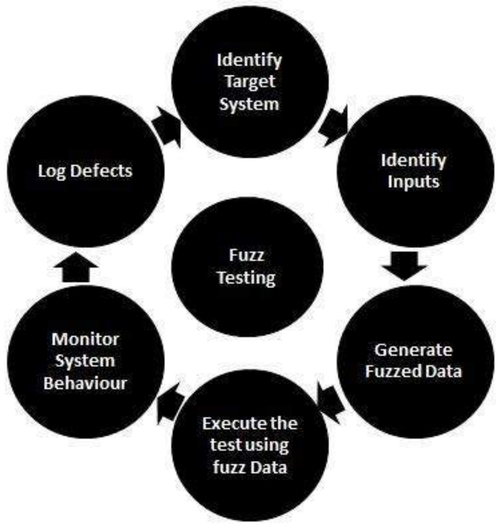


Figure 3: A typical fuzz testing flow diagram [6] depicting how to approach fuzzing. Fuzzing, generally speaking, should use feedback from fuzzing to inform future actions/generated test cases. All test cases (inputs and results/return values) should be logged for later analysis.

3.1 Syscall Fuzzers

Syscall fuzzing is a type of fuzzer that is specifically focused on generating random inputs and sending them to the various routines that make up the interface to the kernel.

Some of the earliest references I've found regarding syscall fuzzing tools date back to 1991. One of the more well-known projects went by the name of `tsys` ("test syscalls") [27] and was composed by Tin Le. This tool was *very* simple (i.e., the alpha release was less than 200 lines of code!) and essentially boiled down to iterating over each of the syscalls (identified by the integer ID) and invoking the `syscall()` routine while trying to throw complete and utter garbage at each one with no apparent regard for things like proper data types or even the correct number of arguments for a given syscall that was selected to be fuzzed. Le was apparently motivated by another project, `crashme` [20], that was originally developed by George Carrette and was also a project that was, according to the (seemingly) official website, "...a tool for testing the robustness of an operating environment using a technique of "Random Input" response analysis," which is, in effect, a fancy way of saying "fuzzing." There have since been many variations of `crashme` carried forward by others (e.g., kg_crashme, ak_crashme, dj_crashme)—similar to tsys, these projects were all fairly naïve fuzzers.

Flash forward some years (roughly the early to mid 2000s) and we begin to see more intelligent implementations of syscall fuzzers. For example, Dave Jones began a project in 2006 that didn't really gain popularity until 2010 when he began investing more in the development of the project and renamed the project to *Trinity* [12]; Trinity seems to be among the most popular open source syscall fuzzers in active development at the time of writing this paper (March 2016). Trinity is discussed in greater detail in section 4.

Another example, XNU Fuzz [5], was a much smaller project that fuzzed only a subset of the syscalls and was primarily focused on finding memory-related issues. XNU Fuzz is a good example of how simple and effective a semi-random fuzzer can be in helping to identify kernel-level bugs.

A similar tool to Trinity, I Know This ("iknowthis") [10], was a syscall fuzzer tool for UNIX-like systems that was developed by Google. Much like Trinity, the goal of iknowthis was to identify unexpected behavior in response to semi-random generated data. According to the code survey I did, and similar to Trinity (as will be described later), iknowthis makes an effort to create legitimate resources (e.g., sockets, file descriptors) that are passed to system calls in order to pass the basic input validation that is normal in kernel system call implementations today (e.g., checks for valid sockets/file descriptors). Unlike Trinity, there seemed to be far less support for the wide-variety of UNIX-like systems, though it appears that iknowthis did have a nice web UI for reporting the fuzz testing results. The iknowthis project, for some reason, is no longer active according to its Google code archive—the most recent commit was November 26, 2012 and stated that it "No longer compiles on recent kernels."

3.2 Other Fuzzers

While this report is largely focused on syscall fuzzing, it is important to note that fuzzing is not exclusive to the syscall interface. In fact, there are many projects and products/services that are relatively active today that target other types of interfaces (e.g., network stacks/protocols, libraries, applications, files).

Sully [15, 25] is an open source, Python-based, easily-extensible fuzz testing framework that seems to be focused on fuzzing remote targets. Sully was, for a long time, regarded as one of the better fuzzing tools that was available due to its simple data representation and easy/improved data transmission and target monitoring capabilities.

American Fuzzy Lop (AFL) [1] is a popular security-oriented fuzzer that is said to employ "a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary." AFL is an instrumentation-based fuzzing tool that can be used to fuzz binaries that consume various file formats as input. The binary (e.g., application), in general, takes a file as input and AFL iteratively manipulates the contents and size of the file in order to fuzz the application and observe its response to the various file manipulations. Other, less popular but still relevant, fuzzers that perform fuzzing on applications that consume file input include: the Basic Fuzzing Framework (BFF) [26], the ZZUF Multi-purpose Fuzzer [18], and Radamsa [28].

Another example, Peach Fuzzer [16], is a modern (unfortunately, commercialized) fuzzer that has a web-based GUI for easier use, and is capable of fuzzing various targets (i.e., not just the syscall interface) and building "aware" models that give the fuzzer knowledge to generate meaningful test cases, generating sophisticated reports about fuzzing results, and can lend helpful insights into identifying known vulnerabilities.

While the purpose of this report is not to cover the entire history of fuzzing, nor to describe any one fuzzer in great detail, it felt appropriate to delineate the fact that fuzzing has a long history and is still a relevant technique today.

3.3 Android Fuzzers

There does not appear to be a significant amount of available/published work in the space of fuzzing (on) Android. What follows is an overview of the various tools and works-done that I did manage to find.

A potentially helpful starting point for people looking into fuzz a particular application running on an Android mobile device via ADB might be interested in the ADBFuzz project [14]. While this project is no longer maintained and admitted limited (or no) support for environments that differed from the author's (which was not particularly well-defined), this project used websockets (over ADB) to fuzz the Mozilla Firefox application. ADBFuzz is primarily implemented in Python.

Another Android fuzzing related project is the *Broadcast Intent Fuzzing Framework for Android* ("bifuz"") [7]. The bifuz project, similar to ADBFuzz, is implemented largely in Python and uses ADB to fuzz some target device and, more specifically, individual packages (i.e., APKs) on that target device; bifuz made it possible to generate random intents which were then sent to the specified packages on the target device.

The most recent work that I was able to find (and indeed I found this so recently that I haven't had enough time to fully explore all of the tools/references to which this work refers) on Android fuzzing was by Alexandru Blanda of Intel; he presented his work [17] at *Black Hat Europe 2015* which detailed his efforts to fuzz specific Android OS components in search of vulnerabilities. Blanda found a number of vulnerabilities in the Stagefright framework, the mediaserver process, the Android APK install process, the installed daemon, dex2oat, and the ART. In his work, Blanda largely describes his utilization of the AFL [1] fuzzer (a version that had been ported to Android) since his work involved manipulating files (e.g., media files, APK files) that are read-in and processed by various Android components. The code specifically relevant to fuzzing the media server was open-sourced under the Media Fuzzing Framework for Android (MFFA) [8].

The fuzzer that I found most promising for our own future exploration of Android fuzzing is the *fuzzer-android* project [11] (a subset of which is a fork of Trinity) by "informationextraction" on GitHub. There is no documentation for the project which has made it *very* difficult to navigate. In Dave Jones' blog (the author of Trinity), he did refer to the fact that the android-fuzzer fork of Trinity was the Hacking Team's attempt at modifying Trinity to fuzz `ioctl()` on Android [19]. The project is known to have yielded some results but no real analysis has been done on the success of such work in producing crashes that are exploitable or already fixed.

4 Linux Syscall Fuzzing with Trinity

Trinity [12] is an open source system call ("syscall") fuzzing application. While syscall fuzzers are not an altogether new concept, Trinity is an exceptional tool in that it implements its fuzzing with special consideration given for the basic requirements of the various Linux syscalls such as argument types, file descriptors, finite ranges of values, and so forth. For example, if a syscall such as `read()` or `write()` is invoked, and that syscall expects a valid file descriptor, Trinity does work to ensure that such a file descriptor exists and will be used when the relevant syscall interface is being exercised. By handling basic input argument validation, Trinity enables testing of "deeper" functionality and can be helpful for identifying bugs that would be harder to target with a more naïve approach to fuzzing. Trinity also provides a wealth of log data that is `fsync()`-ed prior to each syscall that is fuzzed—this is useful in the event that one of the test cases results in a kernel panic. All log-related files are stored in the `tmp/` directory in the base of the Trinity project folder. A slightly dated, but more in-depth analysis of the Trinity fuzz tester was presented by Michael Kerrisk on *LWN.net* [23].

To experiment with Trinity [12] and how it runs I setup a 32-bit Linux virtual machine using VirtualBox and pulled the open source code from GitHub. Once my VM was setup and properly provisioned with relevant tools (e.g., make, git), I ran the following to clone the Trinity project from GitHub, configure it, make it, and run it (Note: various warnings were emitted while building the project, however, there is no indication in the documentation or the observed execution that this caused any problems with the core functionality of Trinity):

```
$ git clone https://github.com/kernelslacker/trinity
$ cd trinity
$ ./configure
$ make
...
$ ./trinity
```

4.1 Trinity options

The documentation for Trinity describes various options that can be passed when running the program. What follows is an overview of those most pertinent to our work:

```
$ ./trinity -L
```

Of particular use to me was running Trinity with the `-L` option which allowed me to see all of the enabled/disabled syscalls. All syscalls were enabled in my environment by default.

```
$ ./trinity -I
```

It was also interesting to review the output of the `-I` option which dumps all of the available ioctls.

```
$ ./trinity -c ioctl
```

Lastly, because we are most interested in stressing a particular syscall in this project (e.g., ioctl), it was helpful to learn that it is possible to use the `-c SYSCALL` option with Trinity to concentrate the fuzzing test cases on a certain syscall.

5 Future Work & Conclusion

In this project I described my findings as I studied Android, its Binder IPC mechanism, and fuzz testing with a special concentration on Linux syscall fuzzers and Android fuzzing. I have no doubt that Android's Binder is ripe for vulnerability exploration due to the limited amount of public knowledge and known vulnerability testing performed against Binder. In future work I hope to work out issues in my test/build environment so that I can actually fuzz the Binder kernel driver—specifically the `ioctl()` handlers—and produce meaningful documentation that may help others get setup to do similar studies.

References

[1] American Fuzzy Lop: Security-oriented Fuzzer. http://lcamtuf.coredump.cx/afl/.

[2] Android Binder: Android Interprocess Communication. https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf.

[3] Binder — Android Developers. https://developer.android.com/reference/android/os/Binder.html.

[4] Binder Data Transfer Overview Diagram. http://www.cubrid.org/.

[5] fintler/xnufuzz: An XNU kernel fuzz tool. https://github.com/fintler/xnufuzz/.

[6] Fuzz Testing Overview Diagram. http://www.tutorialspoint.com/.

[7] fuzzing/bifuz: Broadcast Intent FUZzing Framework for Android. https://github.com/fuzzing/bifuz.

[8] fuzzing/MFFA: Media Fuzzing Framework for Android. https://github.com/fuzzing/MFFA.

[9] IBinder — Android Developers. https://developer.android.com/reference/android/os/IBinder.html.

[10] iknowthis: syscall fuzzer for UNIX-like systems. https://code.google.com/archive/p/iknowthis/.

[11] informationextraction/fuzzer-android: Linux/Android system call fuzzer. https://github.com/informationextraction/fuzzer-android.

[12] kernelslacker/trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[13] Linux 3.19 - Linux Kernel Release Summary. http://kernelnewbies.org/.

[14] mozilla/ADBFuzz: Fuzzing Harness for Firefox Mobile on Android. https://github.com/mozilla/ADBFuzz.

[15] OpenRCE/sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.

[16] Peach Fuzzer. http://www.peachfuzzer.com/.

[17] Alexandru Blanda. Fuzzing Android: a recipe for uncovering vulnerabilities inside system components in Android. https://www.blackhat.com/docs/eu-15/materials/eu-15-Blanda-Fuzzing-Android-A-Recipe-For-Uncovering-Vulnerabilities-Inside-System-Components-In-Android-wp.pdf.

[18] Caca Labs. ZZUF - Multi-purpose Fuzzer. http://caca.zoy.org/wiki/zzuf.

[19] Dave Jones. Future development of Trinity. http://codemonkey.org.uk/2015/07/12/future-trinity/.

[20] George J. Carrette. CRASHME: Random input testing. http://people.delphiforums.com/gjc/crashme.html.

[21] Google. The Android Stack Diagram. http://source.android.com/source/index.html.

[22] Kaladharan, Y., Mateti, P., and Jevitha, K. P. An Encryption Technique to Thwart Android Binder Exploits. In *Intelligent Systems Technologies and Applications*, S. Berretti, S. M. Thampi, and S. Dasgupta, Eds., vol. 385. Springer International Publishing, pp. 13–21.

[23] Michael Kerrisk. LCA: The Trinity fuzz tester. http://lwn.net/Articles/536173/.

[24] Palmsource Inc. OpenBinder documentation. http://www.angryredplanet.com/ hackbod/openbinder/.

[25] Pedram Amini, and Aaron Portnoy. Sulley Manual. http://fuzzing.org/wp-content/SulleyManual.pdf.

[26] The CERT Division. Basic Fuzzing Framework (BFF) — Vulnerability Analysis. https://www.cert.org/vulnerability-analysis/tools/bff.cfm.

[27] Tin Le. TSYS – Google Groups. https://groups.google.com/forum/.

[28] University of OULU. Radamsa. https://www.ee.oulu.fi/research/ouspg/Radamsa.