Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations

Antonio Ken Iannillo*, Roberto Natella*, Domenico Cotroneo*, Cristina Nita-Rotaru†
*Università degli Studi di Napoli Federico II, Naples, Italy †Northeastern University, Boston, USA
*{antonioken.iannillo, roberto.natella, cotroneo}@unina.it †c.nitarotaru@neu.edu

***Abstract*—Android has become the most popular mobile OS, as it enables device manufacturers to introduce customizations to compete with value-added services. However, customizations make the OS less dependable and secure, since they can introduce software flaws. Such flaws can be found by using fuzzing, a popular testing technique among security researchers.**

This paper presents *Chizpurfle*, a novel "gray-box" fuzzing tool for vendor-specific Android services. Testing these services is challenging for existing tools, since vendors do not provide source code and the services cannot be run on a device emulator. *Chizpurfle* has been designed to run on an unmodified Android OS on an actual device. The tool automatically discovers, fuzzes, and profiles proprietary services. This work evaluates the applicability and performance of *Chizpurfle* on the Samsung Galaxy S6 Edge, and discusses software bugs found in privileged vendor services.

***Index Terms*—Android OS; robustness testing; fuzzing; vendor customizations.**

I. Introduction

Android comes in different flavors, depending on which vendor is implementing it. Nowadays, more than 20 original equipment manufacturers (OEMs), including but not limited to Samsung, HTC, Huawei, Motorola, and LG, base their devices on the Android Open Source Project (AOSP). Hardware capabilities are not the only factor that support the customers' choice. Software customizations play a key role in this aspect, making user experience unique and more enjoyable. For example, vendor customizations include services for providing mobile personal assistants [1]–[3], advanced photo enhancement [4], [5], mobile payments [6], *etc*.

Unfortunately, these customizations often introduce new software defects, which are vendor-specific. Because they are proprietary, vendor customizations are not integrated in the open-source Android and do not benefit from the feedback loop of the whole ecosystem. Thus, they are less scrutinized than the core AOSP codebase, and their vulnerabilities take significantly more time to be patched: for example, the Google Android security team publishes a monthly security bulletin [7] with new and patched security vulnerabilities, but it has to refer the users to vendor-specific security bulletins such as the ones by LG [8], Motorola [9], and Samsung [10]. It is worth noting that vendors customizations consist of code running with special privileges, thus exacerbating the security issues[1].

[1]For example, recent devices based on Qualcomm chipsets suffer from a vulnerability in the Qualcomm service API that allows privilege escalation and information disclosure [11].

Fuzzing is a well-established and effective software testing technique to identify weaknesses in fragile software interfaces by injecting invalid and unexpected inputs. Fuzzing was initially conceived as a "black-box" testing technique, using random or grammar-driven inputs [12]. More recently, "white-box" techniques have been leveraging information about the program internals (such as the test coverage) to steer the generation of fuzz inputs, either by instrumenting the source code or by running the target code in a virtual machine [13], [14]. The visibility of the test coverage has dramatically improved the effectiveness of fuzzing tools, as showed by the high number of subtle vulnerabilities found in many large software systems [13], [15], [16]. Unfortunately, these tools are not applicable to proprietary Android services, since vendors are not willing to share their source code, and since virtual machine environments (*e.g.*, device emulators) do not support the execution of these proprietary extensions.

In this paper, we introduce a novel "gray-box" tool, named *Chizpurfle*, to address the gap in the spectrum of mobile fuzzers, and to improve the effectiveness of fuzzing on vendor customizations. Similarly to recent white-box fuzz approaches, Chizpurfle leverages test coverage information, while avoiding the need for recompiling the target code, or executing it in a special environment. The tool has been designed to be deployed and run on unmodified Android devices, including any vendor customization of the Android OS. The tool leverages a combination of dynamic binary instrumentation techniques (such as software breakpoints and just-in-time code rewriting) to obtain information about the block coverage. Moreover, *Chizpurfle* is able to guide fuzz testing only on the vendor customizations, by automatically extracting the list of vendor service interfaces on an Android device. The tool also provides a platform for experimenting with fuzz testing techniques (such as evolutionary algorithms) based on coverage-based feedback.

We validated the applicability and performance of the *Chizpurfle* tool by conducting a fuzz testing campaign on the vendor customizations of the Samsung Galaxy S6 Edge, running Android version 7. We found that *Chizpurfle* improves the depth of testing compared to the black-box approach, by increasing the test coverage by 2.3 times on average and 7.9 times in the best case, with a performance overhead that is comparable to existing dynamic binary instrumentation frameworks. Moreover, we discuss two bugs found in privileged services during these evaluation experiments.

The rest of this paper is structured as follows. Background and motivations for this work are discussed in Section II. Previous related work and tools are discussed in Section III. The design of Chizpurfle is described in Section IV, while its evaluation is presented in Section V. Finally, conclusions and future work conclude the paper in Section VI.

II. Background and Motivation

When a vendor delivers a new smartphone on the market, it includes several customizations of the *vanilla* Android, the open source software stack from the Android Open Source Project (AOSP). Unlike AOSP, customizations are usually closed source and undocumented, and vary among vendors. Vendors' software customizations are focused on three areas:

- *Device drivers*: they support proprietary hardware components of the smartphone;
- *Stock applications*: they are pre-installed on the smartphone along with the default AOSP stock applications;
- *System services*: they enhance the Android OS with additional APIs for both stock and third-party applications.

We focus on the third type of customizations, *i.e.*, system services, because they usually run as privileged processes (thus, they have a major potential impact on robustness and security); they are directly exposed to (potentially buggy and malicious) user applications; they provide wrappers to lower-level interfaces, such as device drivers; and they represent a large part of vendor customizations.

In order to understand the extent of deployment of vendor customizations, we conducted a preliminary analysis of system services from vendor customizations in three commercial smartphones, namely the HTC One M9, the Huawei P8 Lite, and the Samsung Galaxy S6 Edge. We extracted the services interfaces on the three devices and on their corresponding Android AOSP versions, using the same techniques of the *Chizpurfle* tool (that are further discussed in §IV-B), and compared the two lists.

TABLE I reports the results of this analysis. The first row is the version of the Android Platform running on each device. The second row is the number of services found only on the device, but not in the corresponding AOSP; in the third and forth rows, this number is split between Java and C services. The next two rows refer only to the Java-implemented services, of which we could retrieve the methods signatures through Java Reflection. The fifth row considers the common Java services, present in both AOSP and vendor devices, that have new methods in the vendor version. Finally, the last row shows how many new methods are present in the vendor services that do not exist in the AOSP. Our analysis shows that there is a significant number of customized services and vendor-specific methods. Moreover, most of these services execute in the context of privileged processes (such as *system_server*, *media_server*, *etc.*), where any failure can have a severe impact the whole OS.

The large vulnerability surface and high privilege of proprietary services motivate the need for specialized tools to evaluate their robustness. To achieve its full potential, fuzz testing needs to guide the generation of inputs according to test coverage, as demonstrated by empirical experience in several security-critical contexts [13], [15], [16]. However, the lack of source code for proprietary services, and the inability to run these proprietary extensions on a device emulator, defy the strategies for profiling coverage that are adopted by existing fuzzing tools.

TABLE I
Vendors' Smartphone Customizations on System Services

	Huawei P8 Lite	HTC One M9	Samsung Galaxy S6 Edge
Android version	5.0	6.0	7.0
# new services	30	7	82
# new C services	13	2	20
# new Java services	17	5	62
# extended Java services	15	25	52
# new Java methods	325	166	2,272

III. Related Work

This section gives an overview of previous work in the general area of fuzzing, and discusses how *Chizpurfle* improves over existing tools for mobile device fuzzing.

OS and systems software fuzzing. Since its initial years, fuzz testing has been extensively adopted for testing systems software, such as network servers, shell applications, libraries, and OS kernels. The early study by Miller *et al.* on fuzzing UNIX system utilities [12], by injecting random inputs through their command line interface and standard input stream, found a surprisingly high number of targets that experienced crashes, leaks and deadlocks, even when exposed to apparently trivial (but invalid) inputs. Other approaches for OS robustness testing, such as BALLISTA [17], MAFALDA [18], and the DBench project [19] injected invalid inputs by bit-flipping them or replacing them with "difficult" inputs, or forced the failure of kernel APIs and device drivers [20], [21].

Among the most modern and mature fuzzing tools, *American Fuzzy Lop* (AFL) is well-known for having found notable vulnerabilities in dozens of popular libraries and applications [13]. AFL is an "instrumentation-guided genetic fuzzer", which modifies the target program at compile-time in order to efficiently profile the branch coverage during the execution of the tests, and to communicate with the main AFL process. Based on coverage measurements, AFL iteratively improves the quality of fuzz inputs, by mutating the previous inputs that discovered new paths. AFL has also been extended to avoid compile-time instrumentation, by using the QEMU virtual machine to trace the instructions executed by the target (at the cost of higher run-time overhead and of the additional dependency on a virtual machine emulator). Another example of coverage-guided fuzzer is *syzkaller* [22], which also uses QEMU and compile-time instrumentation to fuzz the whole Linux kernel through its system call interface.

Another significant advance has been represented by white-box fuzzing techniques that leverage symbolic execution. The

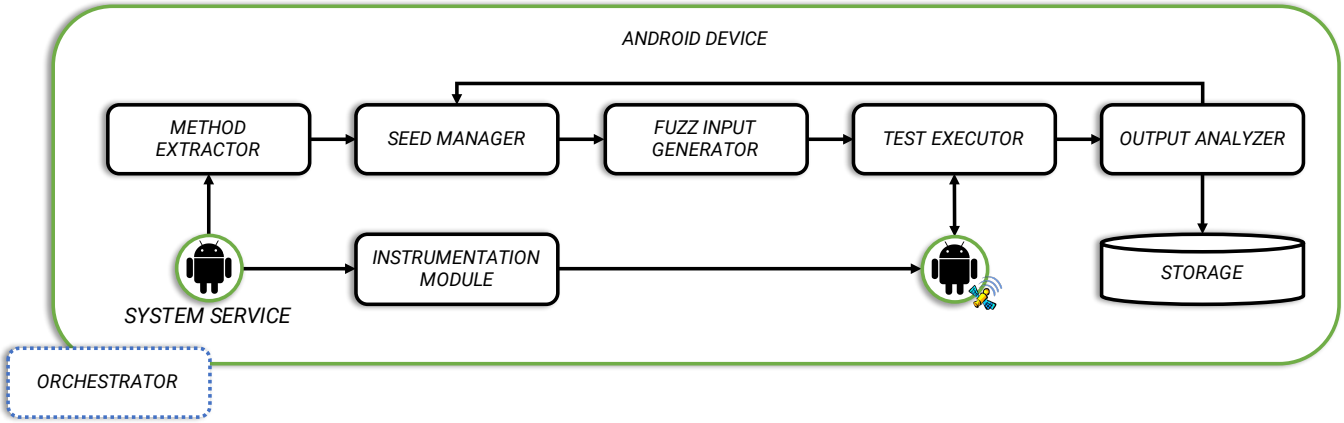

Fig. 1. Overview of the Architecture of Chizpurfle.

most well-known is *KLEE* [14], a virtual machine environment, based on the LLVM compiler infrastructure, with a *symbolic state* for every memory location (*i.e.,* boolean conditions that must hold at a given point of the execution) that is updated as code is executed by an interpreter. When KLEE encounters a branch condition, it forks in two execution flows, each with a different constraint on the variables involved in the branch condition. When a failure path is found, a *constraint solver* is used to find an input that fulfills all the conditions on that path. *SAGE* [23] is another well-known fuzzing tool by Microsoft: starting from some (tentative) concrete input, the tool traces the program execution using a record&replay framework [24] to identify the path constraints for the input; then, it negates one of these constraints, and uses a constraint solver to generate inputs to cover the new conditions. It is important to note that white-box fuzzing is extremely powerful, but very resource-consuming due to the overhead of constraint solving and to the exponential explosion of program paths. Thus, these techniques are best applied in combination with black-box fuzzing: Bounimova *et al.* [15] report a split of 66%-33% of bugs found respectively by black- and white-box fuzzing during the development of Microsoft's Windows 7. Moreover, white-box fuzzing can only be applied when the target is executed in an environment (such as a virtual machine) able to trace and to fork symbolic states.

Android fuzzers. In Android-related research, fuzzing has been extensively used to attack network and inter-process interfaces. For example, Mulliner and Miller [25] found severe vulnerabilities in the SMS protocol. Droidfuzzer [26] targets Android activities that accept MIME data through Intents (a higher-level IPC mechanism based on Binder); Sasnauskas and Regehr [27] developed a more generic Intent fuzzer that can mutate arbitrary fields of Intent objects. Mahmood *et al.* [28] adopted the white-box fuzzing approach by decompiling Android apps to identify interesting inputs and running them on Android emulator instances on the cloud. However, these and similar tools [29]–[32] focus on the robustness of Android apps, and can not be directly applied to fuzz Android system services.

To the best of our knowledge, the few notable studies on fuzzing Android system services are the ones by Cao *et al.* [33] and Feng *et al.* [34]. Cao *et al.* [33] focus on the input validation of Android system services. Their tool, *Buzzer*, sends crafted parcels (i.e., the basic messages on the Binder) to invoke AOSP system services with fuzzed arguments. Since *Buzzer* was an early tool of its kind, it relied on manual efforts for several tasks, such as to identify the arguments of service methods, to avoid fuzzing on methods that could not be invoked by third-party apps anyways (due to limited permissions), *etc.*. Feng *et al.* [34] developed *BinderCracker*, a more sophisticated parameter-aware fuzzer that can automatically understand the format of Binder messages and that supports more complex communication patterns over the Binder (such as callback objects returned by system services). However, both these tools are purely black-box approaches and do not gather any information about the internal coverage of the tested services, thus missing the opportunity to improve the efficiency of fuzzing. This problem has only been partially addressed by Luo *et al.* [35], which recently developed a successor of *Buzzer* that exploits symbolic execution. However, this tool is not applicable to vendor customizations, since it is designed to run outside the Android system and requires the availability of the target source code.

IV. Tool Design

The ***Chizpurfle*** tool architecture is presented in Fig. 1. It includes six software modules running on the target Android device, that are implemented in Java and C. These modules cooperate to profile the target system service and to generate fuzz inputs according to test coverage. We designed *Chizpurfle* to be as less intrusive as possible, and to only require root permissions for few debug operations discussed in this section.

The *Methods Extractor* produces a list of system services and their methods, marking the custom vendor services as described in Section II. It also provides a map between services and their hosting processes. The *Seed Manager* iterates over the custom vendor services and methods, and it provides initial inputs (*seeds*) for testing them. The *Fuzz Input Generator*

takes a seed (either the initial seed, or any previous worthwhile input) and generates new actual inputs for the target method, by applying fuzzing operators to the values of method parameters. Then, the *Test Executor* applies the fuzzed inputs to the target service, while the *Instrumentation Module* keeps track of the test coverage. The outcomes of the test are collected, analyzed, and saved by the *Output Analyzer*. It also provides feedback to the *Seed Manager* with seeds for the next test iteration. Finally, the *Orchestrator* provides a simple user interface for *Chizpurfle*.

A. Orchestrator

The *Orchestrator* is the only part of *Chizpurfle* that runs outside the target Android device (*i.e.,* on the user's workstation), that loads and controls the other modules using the Android Debug Bridge (ADB) [36] through an USB connection. *Chizpurfle* minimizes the amount of interactions through ADB, since this connection is notoriously unstable, and we could not rely on it due to potential side effects of fuzzing. Thus, *Chizpurfle* is detached from the ADB *shell* process right after it is started, in order to avoid any issue related to the ADB connection. Test data are recorded on a local file on the device and later pulled from the target device by the *Orchestrator*; the *Orchestrator* periodically checks the progress of fuzz tests by briefly connecting with ADB and inspecting the logs of *Chizpurfle*.

We also need to prevent the early termination of *Chizpurfle* in the case of crashes of system processes. If *Chizpurfle* ran as a standard Android app, it would be bound to *Zygote*, which is a daemon process that serves as parent for all Android processes, and which provides a pristine copy of the Android Runtime environment for its children through copy-on-write mechanism. When the *Zygote* dies, all children processes die as well. Thus, we run *Chizpurfle* modules in a distinct Android Runtime from the *Zygote*, that is launched by the *app_process* utility (the same utility that starts *Zygote* at boot). This enables *Chizpurfle* to keep working and gather data even if key system processes fail due to vulnerabilities in vendor customizations.

B. Method Extractor

The *Method Extractor* gets the list of services from the Service Manager in a vendor-customized Android device, and it compares them with a blueprint of the AOSP with the same Android version.

The Android OS provides a service-oriented architecture to manage its several services, as shown in Fig. 2. At boot time , ❶ the Service Manager registers itself as the "context manager", by sending a special message to the Binder driver, which is the main inter-process communication mechanism provided by the Linux kernel of Android. Then, ❷ a service provider publishes its services by sending a message through the Binder driver to the Service Manager. When a client application wants to contact a service, ❸ it first queries the Service Manager with the service name, and then ❹ it invokes the service directly through Binder.

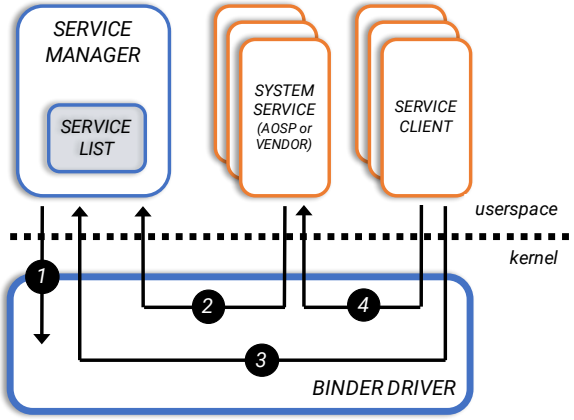

Fig. 2. Android Services and Service Manager.

The *Method Extractor* queries the Service Manager on the target device to get the list of all registered services, including customizations. By iterating on these names, it retrieves the list of service descriptors. In case of Java-implemented services (supported by the current version of the tool), a service descriptor is the string name of the Java Interface that is implemented by that system service (*e.g.,* the *Package Manager* service implements the *android.content.pm.IPackageManager* Java Interface). Then, Java Reflection API is used to inspect the definition of the interfaces, and to get the signatures of the methods in the service. The methods that are not in the AOSP are marked as "vendor customizations" and considered for testing.

Another task of the *Method Extractor* is to map every service to the system process that hosts that service. This mapping is obtained by hooking calls to the Service Manager, before the services are registered. In particular, we focus on invocations of the function:

```
static int
svc_can_register(const uint16_t *name,
        size_t name_len,
        pid_t spid,
        uid_t uid)
```

where *spid* is the PID of the process that wants to register the service named *name*. The functions of Service Manager are hooked by copying a breakpoint handler in the memory address space of the process and by modifying the symbol table to hijack function invocations (the technique to modify the memory of the target process is further discussed in the next subsection about the *Instrumentation Module*). We force the system services to be published again (thus invoking the Service Manager) by restarting the Zygote process, which in turn forces the restart of system processes and their services. If the method returns 1, then the service has been correctly registered, and the *Methods Extractor* retrieves the name of the process and saves the mapping.

C. Instrumentation Module

The *Instrumentation Module* interacts with the process that runs the target service, in order to collect information about the test coverage. We designed the *Instrumentation Module* by taking into account the following requirements: (1) it must be able to intercept the execution of branches by the target service, in order to identify any new code block covered by the test; (2) it has to attach to system processes that are already running, since the life cycle of Android services (including vendors' ones) cannot be directly controlled by external tools such as *Chizpurfle*, and since most of these service are already running since the boot of the target device; and (3) it should be able to instrument proprietary services on the actual device (which is the goal of this study), thus excluding any approach that recompiles the source code or that runs in an emulated environment.

We initially explored both hardware and software solutions to measure coverage. Hardware solutions typically take advantage of special CPU features for debugging purposes, such as performance counters. The ARM processors (the CPU family also adopted in Android devices) provide the CoreSight on-chip trace and debug utility to trace the execution of program [37]. However, this specific feature is not mandatory for ARM CPUs, and it is not available on the CPUs typically used in Android devices. Thus, we could not use the hardware support from the CPU, since this solution could not be applied on commercial devices.

We then focused on software-based solutions, which typically have a higher run-time overhead, but they can also provide more flexibility and have less requirements about the underlying hardware. In particular, we based our design on the *ptrace* system call of the Linux kernel: it allows a debugger process (in our context, the *Instrumentation Module*) to inspect and to write on the memory address space and CPU registers of the debuggee (in our context, the process that runs the target system service). Typically, debugging tools use ptrace to install *software breakpoints*, by replacing an instruction of the debugged program with another instruction that stops the program and triggers a breakpoint handler function.

We leverage the ptrace mechanism to profile the target code through *dynamic binary rewriting*, which is a general technique used by virtual machine interpreters. The program is divided in *basic blocks*, which are small groups of sequential machine instructions that end with a branch. When the exit branch is reached, the control flow is returned to the interpreter, which retrieves the next basic block, applies some transformations (such as just-in-time compilation and instrumenting the final branch instruction) and moves the control flow to the block; or the exit branch directly jumps to the next basic block if it has already been processed and cached. In our context, we apply the same principle to keep track of which code blocks are executed, in order to compute the test coverage.

Fig. 3 shows the instrumentation and tracing mechanism used by *Chizpurfle*. The *Instrumentation Module* injects into

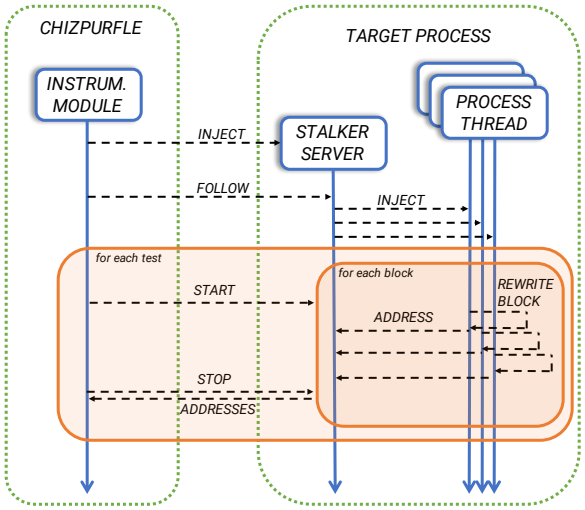

Fig. 3. Chizpurfle Instrumentation and Tracing Mechanism.

the target process a small C library by using ptrace; then, before restoring the execution of the traced process, it starts a new thread in the process to run the library code, which starts the "stalker" server. This server opens a local socket to talk back with the *Instrumentation Module*. At the beginning of a test campaign, *Chizpurfle* sends a message over this socket to enable the tracing of any thread in the target process. Then, the stalker server rewrites the current code block; from this point on, the code blocks will return the control flow to the injected library, which will rewrite the next code block that will be executed by the target. For every rewritten block, the tool adds instructions to log the memory address of the code block, in order to record that the block has been covered. The list of the addresses of covered code blocks is collected by the stalker server in a global data structure. At the end of testing, *Chizpurfle* sends a message to disable logging, and to let the stalker send back to *Chizpurfle* the list of code blocks that have been covered.

In the current version of *Chizpurfle*, we implemented this approach using the Frida framework [38]. Frida is a generic dynamic instrumentation toolkit that provides basic facilities for dynamic binary rewriting, in order to let developers to insert probes in a program for debugging and reverse-engineering purposes. We have ported Frida to 64-bit ARM processors in order to let it run on actual Android devices, and we extended the code rewriting process to trace the coverage of code blocks.

D. Seed Manager

The *Seed Manager* is in charge of providing seeds (*i.e.,* initial inputs for the target service) to the *Fuzz Input Generator*. The *Seed Manager* manages a priority queue of seeds to be fuzzed, which are ordered with respect to their score π. This score is assigned by the *Output Analyzer* (as discussed later in §IV-G), after that the seed has been submitted to the target, and that the coverage for the input has been measured. The score π represents the number of new blocks executed by the

traced process. If π is greater than zero, the seed is fed back to the *Seed Manager* to be further fuzzed in subsequent tests.

This workflow represents the cornerstone for applying evolutionary algorithms to drive fuzz testing towards deeper testing of the target service. To select the next seed from the priority queue, we adopt an exploitation-based constant schedule, where a seed is not used more than once [39]. The termination criterion of *Chizpurfle* is to stop when all seeds have been consumed from the queue, and no more seeds are available for further fuzzing. Moreover, *Chizpurfle* represents a basis for applying several algorithms for fuzz testing, e.g., by changing or tuning the queue scheduling policy and the termination criterion. This is a valuable opportunity for research on fuzzing in mobile devices, as the heuristics and algorithms adopted by existing tools (such as AFL) have evolved over the years on the basis of empirical experience and experimentation with alternative approaches, which is facilitated by tools such as *Chizpurfle*.

At the beginning of a fuzz testing campaign for a target method, the *Seed Manager* creates a new initial seed with empty (for primitive types) or null (for object types) values. This initial seed is not mutated, but immediately submitted as test input. This input will trigger the target method to cover an initial set of π code blocks; then, the input is immediately fed back to the *Seed Manager* to be used as first actual seed with score π. The steps to fuzz a vendor service method are summarized in Algorithm 1.

Algorithm 1 fuzzing a vendor service method

```
Input: Service s, Method m, Process pid
parameters = createInitialSeed(s, m)
outputs = executeTest(s, m, parameters, pid)
analyzedOutputs = analyzeAndSave(outputs)
priorityQueue = {}
priorityQueue.push(parameters, analyzedOutputs.π)
repeat
    parameters, π= priorityQueue.pop()
    for i = 1 to π  do
        parameters' = mutate(parameters)
        outputs = executeTest(s, m, parameters', pid)
        analyzedOutputs = analyzeAndSave(outputs)
        if analyzedOutputs.π > 0 then
            priorityQueue.push(parameters',
            analyzedOutputs.π)
        end if
    end for
until priorityQueue == {}
```

E. Fuzz Input Generator

The *Fuzz Input Generator* receives a seed to be mutated, and generates inputs for the *Test Executor*. Several inputs are obtained from the same seed, by applying different fuzz operators. The number of new inputs to generate is proportional to the score π of the seed, and the fuzz operators are selected according to the types of the parameters of the target method. We implemented in *Chizpurfle* a rich library of fuzz operators, including operators that are often adopted in existing fuzzing tools (including the ones in Section III). For each parameter type, the fuzz operators are:

- *Primitive types* (boolean, byte, char, double, float, integer, long, short): substitute with a random value, substitute with the additive identity (0), substitute with the multiplicative identity (1), substitute with the maximum value, substitute with the minimum value, add a random delta, subtract a random delta, substitute with a special character (only for char);
- *Strings*: substitute with a random string, substitute with a very long random string, truncate string, add random substring, remove random substring, substitute random character from string with special character, substitute with empty string, substitute with null;
- *Arrays* and *Lists*: substitute with array of random length and items, remove random items, add random items, apply fuzz operator on a item value according to its type, substitute with empty array, substitute with null;
- *Objects*: substitute with null, invoke constructor with random parameters, apply fuzz operator on a field value according to its type.

For *Object* types, the *Fuzz Input Generator* provides additional ad-hoc fuzzers for important specific classes defined by the Android OS. For example, the *android.content.Intent* class has a specific fuzzer that injects into the fields of an Intent (such as actions, categories and extras) special values that have a meaning for the Intent (*e.g.,* ACTION_MAIN and ACTION_CALL for the Intent actions) [40]; and the fuzzer for the *android.content.ComponentName* class takes into account which components are installed on the target device, in order to use and to mutate valid component names during fuzz testing. For all the other classes, a generic object fuzzer uses the Java Reflection API to create new objects using the class constructor with random parameters, and to invoke *setter* methods of the class to place random values in the fields of the object.

The *Fuzz Input Generator* keeps a list of all the inputs generated so far, in order not to submit again the same input to the test executor. Seeds are mutated by using a random number generator to select fuzz operators and to guide them (*e.g.,* new values replacing the previous ones are selected randomly). These probabilities are tunable using a configuration file.

F. Test Executor

The *Test Executor* performs tests on the Android device, by invoking the service method with the input provided by the *Fuzz Input Generator*. It generates a proxy for that service using the *IBinderObject* associated to the target service. Before invoking the target method, it flushes the logs collected by the Android OS (the *logcat*, which is a global collector for log messages produced both by user applications and system processes [41]). Then, Chizpurfle sends the start message to the stalker server in the target process (§IV-C) and calls the target method. Any potential exception thrown by the service

is caught, so that the Test Execution is not aborted in the case of service failures. After the method call, it sends another message to the stalker to stop the tracing, and retrieves logs from the logcat. The steps of the *Test Executor* are summarized in Algorithm 2.

Algorithm 2 execute test

Input: Service s, Method m, Parameters p, Process pid
Output: Outputs o

```
1: flushLogcat()
2: startBranchTracing(pid)
3: try: call(s, m, p)
4:     catch e: o.setException(e)
5: o.branches = stopBranchTracing()
6: o.logs = stopLogcat()
```

G. Output Analyzer

The *Output Analyzer* parses the outputs produced by the *Test Executor*, and stores the information and results of the tests on a file on the target device.

This component analyzes the logs to identify any failure that has been triggered by the fuzzing test. A failure is detected using the following criteria:

- **A/F messages**: the system generates log messages with a high-severity level (either *assert* (A) or *fatal* (F)) [41], [42]; such messages are never generated in failure-free conditions, and should be considered as failure symptoms;
- **ANR messages**: the system generates a log message that reports an *ANR* condition (i.e., Application Not Responding) [43]; this condition denotes that the fuzzed input from the *Test Executor* propagated and triggered a long-running operation or an indefinite wait on the main thread of some process;
- **FATAL messages**: the system logs a message reporting a "FATAL EXCEPTION", which denotes an uncaught exception on the service side.

It must be noted that we focus on errors logged by system processes rather than the *Test Executor*; since the *Test Executor* stimulates the system service with invalid input, it is correct for the service to raise exceptions and not to provide any service to the *Test Executor*. Thus, we do not consider these exceptions as failure symptoms as they indicate the correct handling of wrong inputs.

Another check for failure detection is made when the *Test Executor* retrieves the Binder proxy for the tested service (§IV-F). *Chizpurfle* registers a callback, using the *linkToDeath* of the *IBinder* API for the service [44], to receive a notification if the Binder object of the service is not available. This happens when the process that hosts the target service dies.

The *Output Analyzer* component also analyzes the list of block addresses reported by the *Instrumentation Module*. It keeps trace of all blocks covered by tests so far, and compares them with the block addresses of the current test. If new blocks are detected, the test input is assigned a score π, and the new blocks are added to the list of covered blocks.

The outcomes of this analysis, along with general information about the test inputs and the tested service, are saved on a file. If the input receives a non-zero π score, the input is sent to the *Seed Manager* for the next iteration of the fuzzing loop. The steps of the *Output Analyser* are summarized in Algorithm 3.

Algorithm 3 analyze and save results

Input: Outputs o, DeathRecipient r
Output: AnalyzedOuput ao

```
1: ao = o
2: if ("FATAL" or "ANR" in ao.logs.message) or ("F" or
   "A" in ao.logs.level) then
3:     ao.hasFailures = true
4: end if
5: if ao.deathRecipient.deathNotified then
6:     ao.serviceDead = true
7: end if
8: newBranches = ao.branches \ getExecutedBranches()
9: if size (newBranches) > 0 then
10:    addExecutedBranches (newBranches)
11:    ao.π = size (newBranches)
12: end if
13: saveToFile(ao)
```

H. Further Optmizations

When we initially applied the *Chizpurfle* tool to the Samsung Galaxy S6 Edge, we needed to address an important technical problem: the system services (including the ones from vendors' customizations) execute in the context of a few system processes, along with dozens of other threads, such as the *system_server* process, which contains about 160 threads. Unfortunately, instrumenting all these threads at the same time causes a high overhead, that would slow down the execution of the fuzz tests.

We enabled *Chizpurfle* to avoid instrumenting threads that are unrelated to the target service being tested. We base this approach on a simple, yet effective heuristic to detect unrelated threads: for all the services running in the context of the same process of the target service, we tokenize the name of the service, and retain the tokens that belong only to that specific service (for example, in the case of *CocktailBarService*, we retain the tokens "Cocktail" and "Bar"); then, we get the names of the threads of the process, using the *comm* entry in the *proc* file system; finally, we identify the threads whose name include the tokens of services different that the one under testing (for example, we exclude the "*CocktailBarVisi*" thread when testing services different than the *CocktailBarService*). The associations between threads and services can be easily reviewed by *Chizpurfle*'s users before starting the testing campaign. This heuristic reduces the run-time overhead of the instrumentation and only avoids threads that are likely unrelated to the service under testing.

We did another minor optimization to avoid few false positives that happened during the tests. During our preliminary tests, some false positives occurred when the Android device reached a low battery level, that caused the Android OS to switch to battery-saver mode. This change, together with the workload of fuzz tests, slowed down the smartphone, and caused spurious ANRs in processes not related to the service under testing. We prevented these false positives by periodically checking the battery level and pausing the tests if the level is too low. We carefully checked and reproduced all the other failures described in next sections, to assure that our results are free from false positives.

V. Evaluation Campaign

We applied *Chizpurfle* to a well-known commercial smartphone, the Samsung Galaxy S6 Edge. Before testing, we updated this device with the most recent Android OS officially released by Samsung based on Android 7 "Nougat". First, we perform a fuzz testing campaign on all the service methods introduced by Samsung. Then, we perform additional tests to evaluate the performance overhead and the test coverage, compared to a pure black-box approach.

A. Bugs in Samsung Customizations

Chizpurfle detected 2,272 service methods from Samsung customizations. In this first experimental campaign, *Chizpurfle* performed 34,645 tests on these methods. The tool reported that 9 tests caused failures, which are summarized in TABLE II. We executed again the tests, and we found that the failures were reproducible. Then, we analyzed the failure messages reported on the logs, which include uncaught exceptions and the stack trace at the time of the failures. Despite the source code not being available, we notice that the failures affected high-privilege system processes, and were caused by 2 distinct bugs (respectively, the first 4 failures, and the other 5 failures).

The first bug was found in the service *spengestureservice*, hosted by the *system_server* process. The bug was triggered by the method *injectInputEvent*. To understand the role of this method, we analyzed the AOSP, and found a similar method (with the same name and minor differences in the method signature) provided by the *InputManager* class of AOSP, which handles input devices such as keyboards. This method "injects an input event into the event system on behalf of an application" [45]. It is likely that the method with the same name in the *spengestureservice* performs the same operation for input events from the "S Pen" in Samsung devices [46].

One of the input parameters for this method is an array of *android.view.InputEvent* objects, which is an abstract class for representing input events from hardware components. During the fuzz testing campaign, Chizpurfle detected a *FATAL EXCEPTION* when this array is non-null and non-empty, and at least one of its elements is null (instead, the service does not fail if the array is simply null or empty). This input causes the service to throw a *NullPointerException* that is not caught, causing a crash. We found that this bug is fully reproducible. The bug can have two different effects on the Android OS, depending on which process will consume the injected events from the Input Manager. If the events are consumed by the process *com.android.systemui*, the uncaught exception triggers the restart of the process, and a black screen of the user interface for a few seconds. If the events are consumed by *android.ui*, which is a thread of the *system_server* process, the fuzzed inputs has a higher impact: it crashes the *system_server* and causes a restart of the whole Android device. Several method calls with exactly the same parameters values can be arbitrarily managed in both ways.

The second bug was triggered up when fuzzing the method *callInVoIP* of the Samsung's *voip* service. The method likely is used to place a call with *Samsung WE VoIP* app [47], a stock application that provides voice-over-IP for corporate users. The method takes as input parameter a string that represents a SIP address URI (such as "sip:1-999-123-4567@voip-provider.example.net"). *Chizpurfle* found that input strings that include specific SQL control expressions (similarly to single quotes in SQL injection) trigger an uncaught *SQLLiteException* by the *com.samsung.android.incallui* process. This process is a customized version of the *com.android.incallui* process of the AOSP, which handles the UI that appears during a call, providing several on-screen functions. The uncaught exception crashes the *com.samsung.android.incallui* process, cutting off any ongoing call.

B. Comparison with Black-Box Fuzzing

We compared *Chizpurfle* with the black-box approach, to provide a baseline for evaluating our gray-box approach. We first analyze the performance overhead of *Chizpurfle*, that is, the relative slow-down of fuzz testing when applying the gray-box approach. The overhead includes the time for generating inputs and profiling the coverage of the tests. During the whole test campaign on the Samsung Galaxy S6 Edge, *Chizpurfle* measured the overall time spent for executing the test. An individual test takes on average 6.65 seconds, while testing a whole method takes on average 527.60 seconds.

To get the test duration that would be obtained with black-box fuzzing, we performed a second round of tests by disabling both the *Chizpurfle*'s *Seed Manager* and *Instrumentation Module* (the two distinctive elements of gray-box testing). This usage mode *Chizpurfle* (denoted as Chizpurfle$^{\mathrm{BB}}$) is equivalent to perform black-box fuzzing, without neither collecting coverage nor using coverage for selecting the test inputs. In Chizpurfle$^{\mathrm{BB}}$, the inputs are instead generated randomly. For each target method, we used Chizpurfle$^{\mathrm{BB}}$ by applying the same number of inputs that were also generated by the gray-box *Chizpurfle* for that method.

By comparing the time to run Chizpurfle$^{\mathrm{BB}}$ with the time to run the gray-box *Chizpurfle*, we obtain a performance slow-down per service of 11.97x on average. To put this number into context, we must consider that the performance slow-down is inline with other tools for dynamic binary instrumentation. For example the Valgrind framework (which also uses dynamic binary rewriting for complex analyses, such as finding memory

TABLE II
DETECTED FAILURES IN EVALUATION CAMPAIGN

		TESTID	INPUT	FAILURE
spengestureservice	injectInputEvent	7	{0, -2147483648, *array of* android.view.InputEvent *objects with a null item*, false, NULL}	FATAL EXCEPTION: mainProcess: com.android.systemui, PID: 12884 java.lang.NullPointerException: Attempt to invoke virtual method 'long android.view.InputEvent.getEventTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDispatcher.dispatchInputEventInjection (SmartClipRemoteRequestDispatcher.java:201)[...]
spengestureservice	injectInputEvent	22	{-715676118, -1, *array of* android.view.InputEvent *objects with a null item*, false, NULL}	FATAL EXCEPTION: mainProcess: com.android.systemui, PID: 4025 java.lang.NullPointerException: Attempt to invoke virtual method 'long android.view.InputEvent.getEventTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDispatcher.dispatchInputEventInjection (SmartClipRemoteRequestDispatcher.java:201)[...]
spengestureservice	injectInputEvent	162	{0, 91, *array of* android.view.InputEvent *objects with a null item*, false, NULL}	!@*** FATAL EXCEPTION IN SYSTEM PROCESS: android.ui java.lang.NullPointerException: Attempt to invoke virtual method 'long android.view.InputEvent.getEventTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDispatcher.dispatchInputEventInjection (SmartClipRemoteRequestDispatcher.java:201)[...]
spengestureservice	injectInputEvent	186	{-188, 91, *array of* android.view.InputEvent *objects with a null item*, true, NULL}	!@*** FATAL EXCEPTION IN SYSTEM PROCESS: android.ui java.lang.NullPointerException: Attempt to invoke virtual method 'long android.view.InputEvent.getEventTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDispatcher.dispatchInputEventInjection (SmartClipRemoteRequestDispatcher.java:201)[...]
voip	callInVoIP	54	{??9??\u001a??b\u0004A\"1??HanI???\u0017??!\u0014?\u001a\u0006?Fu??UN?\u0015Q??_?\/??\u0007#\u001aX?\u0012?L‘??6}-f??fc??$\u0001?8$s5p?OTg?}??Wu??1=?]?W\u0019␣␣␣␣?z?\u0011?Q??}	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 23452 android.database.sqlite.SQLiteException: near \",\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='\u000e?? [...]
voip	callInVoIP	55	{??_??\u0010␣>\u0001\bK)?}?t'??R?G}T<T\u0001?\u001b?????N?d?V??Z\u0002?e????O??#?\u001dS??\r?????g\u0016\u0002?\u0002?ed\␣␣␣␣u0010?}	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 24643 android.database.sqlite.SQLiteException: near \"???\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='001?\u0 [...]
voip	callInVoIP	72	{??y\u0014?~?\u0011??E\u0007\u000b?'?%?\u0016yD\u0018??9t?i\u000f?NO?6~z␣␣␣␣??Q?(\u0018??\u0002??)\u0003?n?~-?\u0017??^b?\u0015\u00034\u0015PX\u0001?!?G\u0002?\u00␣␣␣␣00}?_??z??v{?A\"Z5?'v?)??f??\u0006n6?j9}	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 25500 android.database.sqlite.SQLiteException: unrecognized token: \"'??9??\u001a????\b?VN6g?,^6\u0011???Lx?\" (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='??9??\u0 [...]
voip	callInVoIP	86	{??o??\bF?%?\u0003?#,??t\u001a??9?^??Z$??J\u0016?\u0011\u0018?\u0016p\u0␣␣␣␣011x\u001c?\u001dTa0␣U'?h?3?????\}	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 32445 android.database.sqlite.SQLiteException: near \"???\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='?\u0011 [...]
voip	callInVoIP	105	{7??L6?I?{<81>?P!:?\u0005?\/?G^\u0003?#\u0000??+c\u0016?\u001eA2??|\f???␣␣␣␣1f@??i-:?~??1\u000f\u001dWSjm???;???\u000f?\u0019\u000f??N[?\u001fWV}	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 5745 android.database.sqlite.SQLiteException: near \"@?d???\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='?0Q?@b}W\u000e [...]

leaks and race conditions), when applied on the SPEC CPU 2006 benchmark [48], causes an average slow-down of 4.3x when the program is simply executed on the Valgrind virtual machine; and an average slow-down of 22.1x when performing memory leak analysis. Such overhead when running tests is rewarded by a higher bug-finding power, and it is in many cases accepted by developers as shown by the widespread adoption of Valgrind in automated regression test suites in open-source projects [49]. In our context, the slow-down still allows the Android system to execute without any noticeable side effect, thus preserving the intended behavior of the test cases. Fig. 4 shows the performance overhead for the two services previously discussed (*voip* and *spengestureservice*), and for other 10 randomly-chosen custom vendor services, which cover the 10% of all the custom methods.

We then evaluate the gain, in terms of test coverage (the higher, the better), obtained by applying gray-box fuzzing instead of black-box fuzzing, given the same time budget T available for both forms of fuzz testing. To measure the test coverage of black-box fuzzing on the vendor customization,

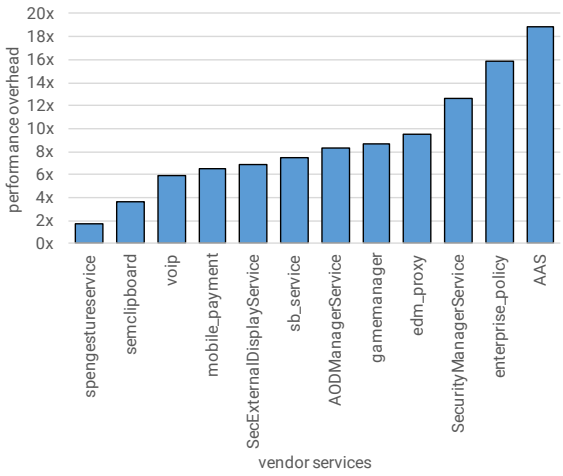

Fig. 4. Performance Overhead of Chizpurfle.

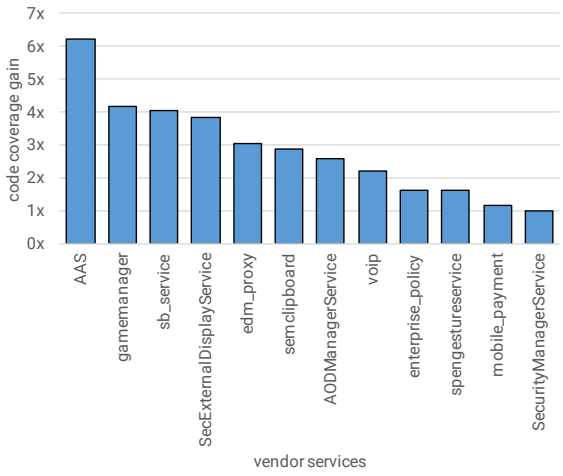

Fig. 5. Code Coverage Gain of Chizpurfle.

the only possible approach is to apply the *Instrumentation Module* of *Chizpurfle* (but without using the *Seed Manager*, in order to fuzz inputs in a random way). We denote this mode as Chizpurfle$^{\texttt{BB+COV}}$.

However, we need to take into account that code instrumentation slows down the execution of the black-box tests, and thus simply applying Chizpurfle$^{\texttt{BB+COV}}$ for the same amount of wall-clock time of the gray-box *Chizpurfle* would unfairly penalize the black-box approach. Therefore, to obtain a fair estimate of the test coverage for black-box fuzzing, we compensate for the slow-down due to instrumentation by granting it a higher time budget than gray-box fuzzing. The time budget is obtained by multiplying the time budget of gray-box fuzzing for the slow-down due to instrumentation (while 11.97x is the average slow-down according to the experiments discussed above, here we applied to each method its slow-down factor).

On average, *Chizpurfle* covers 2.3x more code than the black box approach. The gain in terms of test coverage is shown in Fig. 5 (which focuses on the same services analyzed in Fig. 4). By looking at the code coverage gain per method (see Fig. 6), we noticed that *Chizpurfle* was more effective on those methods that take complex data in inputs, such as *semclipboard*'s method *updateFilter* takes as input an object of type *android.sec.clipboard.IClipboardDataPasteEvent* for managing clipboard data. Instead, in the case of simpler methods, such as getters and setters, the gray-box approach has a minor impact on test coverage.

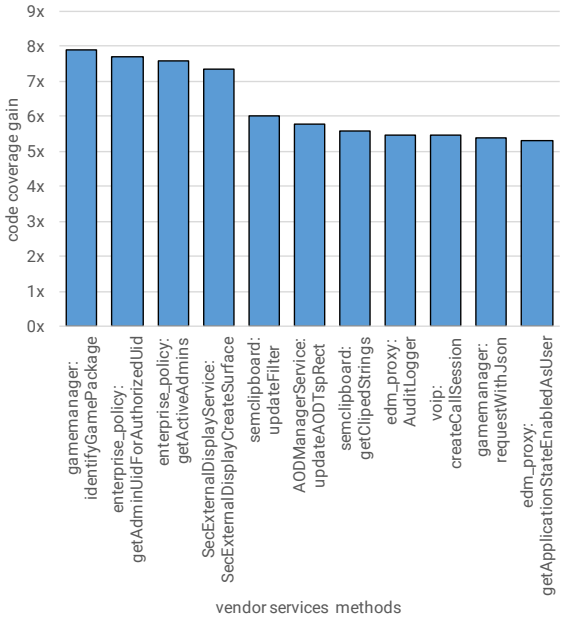

Fig. 6. Code Coverage Gain of Chizpurfle per Method.

VI. Conclusions

This paper presented *Chizpurfle*, a novel gray-box fuzzer designed to test custom system services from Android vendors. This tool exploits dynamic binary instrumentation to measure test coverage and to drive the selection of fuzz inputs. The experimental results on a commercial Android device from Samsung showed that the gray-box approach can discover relevant bugs, that it has a reasonable overhead, and that it can increase the test coverage compared to the black-box approach.

The gray-box fuzzing represents a promising approach for testing proprietary Android services in more depth. The *Chizpurfle* tool represents a valuable opportunity for research on fuzzing in mobile devices, by allowing to experiment with different heuristics for evolutionary fuzzing (e.g., for determining when to stop fuzzing, for prioritizing seeds, and for selecting fuzz operators), as happened for similar fuzzing tools that were applied in different context than mobile devices. Another possible extension of *Chizpurfle* is to include support for system services implemented in C; since there is not reflection API, other reverse engineering techniques should be used in order to extract the method signatures.

ACKNOWLEDGMENT

This work has been supported by UniNA and Compagnia di San Paolo in the frame of Programme STAR (project FIDASTE), and by the COSMIC project (DIETI department).

REFERENCES

[1] Samsung, "Bixby," May 2017. [Online]. Available: http://www.samsung.com/global/galaxy/apps/bixby/

[2] HTC, "HTC U Ultra- HTC Sense Companion," May 2017. [Online]. Available: http://www.htc.com/us/smartphones/htc-u-ultra/#SenseCompanion

[3] Motorola, "Moto Enhancements," May 2017. [Online]. Available: https://www.motorola.co.uk/products/moto-z#moto-enhancements

[4] Huawei, "Huawei P9 co-engineered with leica reinvent smartphone photography," May 2017. [Online]. Available: http://consumer.huawei.com/en/mobile-phones/p9/index.htm

[5] LG, "LG X Cam," May 2017. [Online]. Available: http://www.lg.com/uk/mobile-phones/lg-K580

[6] Samsung, "Samsung Pay," May 2017. [Online]. Available: http://www.samsung.com/us/samsung-pay/

[7] Android, "Android Security Bulletin," May 2017. [Online]. Available: https://source.android.com/security/bulletin/

[8] LG, "LG Security Bulletins," March 2017. [Online]. Available: https://lgsecurity.lge.com/security_updates.html

[9] Motorola, "Moto Security Updates," March 2017. [Online]. Available: https://motorola-global-portal.custhelp.com/app/software-upgrade-security/g_id/5593

[10] Samsung, "Samsung Android Security Updates," March 2017. [Online]. Available: http://security.samsungmobile.com/smrupdate.html

[11] Common Vulnerability and Eposures, "CVE-2016-2060," May 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2060

[12] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[13] Michal Zalewski, "American Fuzzy Lop (AFL)," December 2016. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[14] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[15] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proc. 2013 Intl. Conference on Software Engineering*.

[16] Google Inc., "OSS-Fuzz - Continuous Fuzzing for Open Source Software," 2017. [Online]. Available: https://github.com/google/oss-fuzz

[17] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, 2000.

[18] J.-C. Fabre, F. Salles, M. R. Moreno, and J. Arlat, "Assessment of COTS microkernels by fault injection," in *Proc. Dependable Computing for Critical Applications 7*, 1999.

[19] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau, "Benchmarking the dependability of Windows and Linux using PostMark workloads," in *Proc. 16th IEEE Intl. Symp. on Software Reliability Engineering*.

[20] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations," in *Proc. 33rd Intl. Conference on Software Engineering*. ACM, 2011.

[21] D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella, "Sabrine: State-based robustness testing of operating systems," in *Proc. IEEE/ACM 28th Intl. Conf. Automated Software Engineering (ASE)*.

[22] Google, "syzkaller - linux syscall fuzzer," May 2017. [Online]. Available: https://github.com/google/syzkaller

[23] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008.

[24] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proc. 2nd Intl. conference on Virtual Execution Environments*. ACM, 2006.

[25] C. Mulliner and C. Miller, "Fuzzing the Phone in your Phone," *Black Hat USA, June*, 2009.

[26] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proc. Intl. Conference on Advances in Mobile Computing & Multimedia*, 2013.

[27] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting Intents of Death," in *Proc. Joint Intl. Wksp. on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014.

[28] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *Proc. 7th Intl. Wksp. Automation of Software Test (AST)*. IEEE.

[29] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An Empirical Study of the Robustness of Inter-Component Communication in Android," in *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*.

[30] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android Permission Specification," in *Proc. ACM Conf. on Computer and Communications Security*, 2012.

[31] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: detecting capability leaks of android applications," in *Proc. 9th ACM Symp. on Information, Computer and Communications Security*, 2014.

[32] Y. Hu and I. Neamtiu, "Fuzzy and cross-app replay for smartphone apps," in *Proc. 11th Intl. Wksp. Automation of Software Test*. ACM, 2016.

[33] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services," in *Proc. 31st Annual Computer Security Applications Conf.* ACM, 2015.

[34] H. Feng and K. G. Shin, "Understanding and defending the Binder attack surface in Android," in *Proc. 32nd Annual Conf. on Computer Security Applications*. ACM, 2016.

[35] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "Context-aware System Service Call-oriented Symbolic Execution of Android Framework with Application to Exploit Generation," *arXiv preprint arXiv:1611.00837*, 2016.

[36] Android Studio, "Android Debug Bridge," April 2017. [Online]. Available: https://developer.android.com/studio/command-line/adb.html

[37] ARM, "CoreSight on-chip trace and debug," May 2017. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.coresight/index.html

[38] Ole André V. Ravnås, "FЯIDA," February 2017. [Online]. Available: https://www.frida.re

[39] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[40] AndroidXRef, "Cross Reference: Intent.java," May 2017. [Online]. Available: http://androidxref.com/7.0.0_r1/xref/frameworks/base/core/java/android/content/Intent.java

[41] Android Developers, "Logcat Command-line Tool," May 2017. [Online]. Available: https://developer.android.com/studio/command-line/logcat.html

[42] ——, "Write and View Logs with Logcat," May 2017. [Online]. Available: https://developer.android.com/studio/debug/am-logcat.html

[43] ——, "Keeping Your App Responsive," May 2017. [Online]. Available: https://developer.android.com/training/articles/perf-anr.html

[44] AndroidXRef, "Cross Reference: IBinder.java - linkToDeath," May 2017. [Online]. Available: http://androidxref.com/7.0.0_r1/xref/frameworks/base/core/java/android/os/IBinder.java#257

[45] AndroidXRef, "Cross Reference: InputManager.java - injectInputEvent," May 2017. [Online]. Available: http://androidxref.com/7.0.0_r1/xref/frameworks/base/core/java/android/hardware/input/InputManager.java#833

[46] Samsung, "What are the advantages of S Pen," May 2017. [Online]. Available: http://www.samsung.com/global/galaxy/what-is/s-pen/

[47] ——, "WE VoIP Application for Business," May 2017. [Online]. Available: http://www.samsung.com/us/business/business-communication-systems/unified-communication-solutions/IPX-LSMP/STD

[48] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007.

[49] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault Triggers in Open-Source Software: An Experience Report," in *Proc. 24th Intl. Symp. Software Reliability Engineering (ISSRE)*.