Fuzzing IPC with Knowledge Inference

Kun Yang*†, Hanqing Zhao‡, Chao Zhang*†✉, Jianwei Zhuge*†, Haixin Duan*†
*Institute for Network Sciences and Cyberspace, Tsinghua University
†Beijing National Research Center for Information Science and Technology
‡Chaitin Security Research Lab

***Abstract*—Sandboxing provides a strong security guarantee for applications, by isolating untrusted code into separated compartments. Untrusted code could only use IPC (inter-process communication) to launch sensitive actions, which are implemented in trusted (and maybe privileged) code. IPC-related security bugs in trusted code could facilitate jailbreaks of sandboxing, and thus are becoming high-value targets. However, finding vulnerabilities that could be triggered by IPC is challenging, due to the fact that IPC communication is stateful and format-sensitive.**

In this paper, we propose a new fuzzing solution to discover IPC bugs in IPC services without source code, by combining static analysis and dynamic analysis. We use static analysis to recognize format checks and help construct IPC messages of valid formats. We then use dynamic analysis to infer the constraints between IPC messages, and model the stateful logic with a probability matrix. Therefore, we are able to generate high-quality IPC messages to test IPC services, and discover deep and complex IPC bugs. Without loss of generality, we implemented a prototype MACHFUZZER, for a specific complicated and crucial IPC service, i.e., WindowServer in macOS. This prototype helps us find 12 previously unknown vulnerabilities in WindowServer in 48 hours. Among them, three vulnerabilities are confirmed exploitable, and could be exploited to escape the sandbox and gain root privilege.

***Index Terms*—IPC, Fuzzing, macOS**

I. INTRODUCTION

The code base of modern applications is getting larger and larger. Inevitably, they are prone to have vulnerabilities, e.g., memory corruption bugs which could enable control flow hijacking attacks. Defenders cannot eliminate all vulnerabilities in an application or stop all potential attacks against it. Applications often get compromised.

Sandboxing is a promising solution to stop compromised applications from causing further damages to the system. In general, it isolates untrusted or vulnerable code of target applications into confined processes, and drops all sensitive or privileged permissions of such processes. For example, it could enforce that no subprocess could get spawned inside the sandboxed process. As a result, even in the case that the vulnerable process is compromised, attackers cannot perform sensitive actions. Due to its effectiveness, sandboxing is adopted by more and more modern applications. For example, Google implemented a sandbox for Chrome, which is later used by Adobe Reader. Safari has a sandbox built on top of macOS's mach IPC mechanism.

Jailbreaking is critical for attackers who intend to perform sensitive actions. Note that, the sandboxed process could communicate with other processes (e.g., services with trusted code) via IPC (inter-process communication). Therefore, to escape the sandbox, one of the most common ways used by a compromised sandboxed process is exploiting vulnerabilities in an IPC counterpart unsandboxed process to launch sensitive actions, as shown in many previous exploits [1]. Figure 1 illustrates a classic macOS sandbox escape.

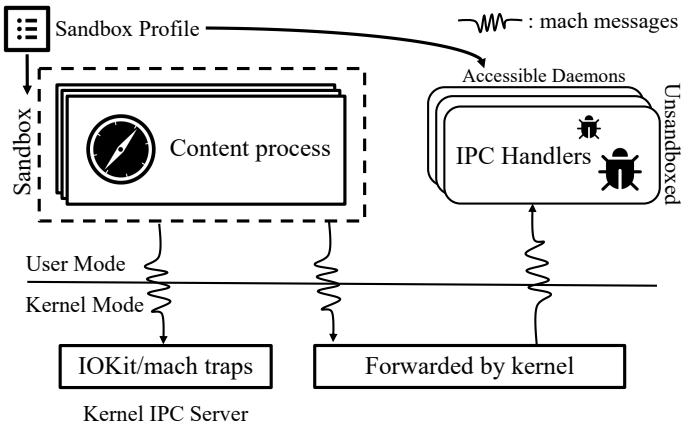

Fig. 1. Illustration of a classic macOS sandbox escape, which exploits vulnerabilities in the unsandboxed services via a popular macOS IPC mechanism, i.e., mach IPC.

So, IPC services that are unsandboxed and privileged have become the everlasting attack surfaces. Finding IPC-related vulnerabilities in such services is thus crucial to the security of the sandbox, both for attackers and defenders.

Currently, fuzzing is the best practice for finding IPC vulnerabilities, which involves repeatedly sending an arbitrary sequence of randomly generated IPC messages. Many security experts and researchers have proposed different fuzzers for different IPC mechanisms or targets. The fuzzer [2] leverage coverage-guided fuzzing to test the Android Binder IPC. Another fuzzer [3] specifically targets parcelization in Android Binder IPC. There are also some researches on fuzzing Intent, i.e., a high-level messaging object used in IPC between Android applications, to detect vulnerabilities that could cause logical permission leaks [4]. RPC (remote procedure call) is the IPC mechanism used in Windows. Back to 2006, iSEC published a fuzzing tool [5] to test Windows RPC. It utilized a straightforward and random-fashion method, which hooks handlers of named pipes and then changes the values randomly. Ret2 Systems [6] proposed a passive fuzzer to test the Mach IPC mechanism in macOS. By using Frida DBI (dynamic binary instrumentation) framework, this fuzzer hooks the target process and injects bitflips at runtime into the hijacked communication channel.

However, existing fuzzing solutions either work in a random way or fail to handle IPC-specific challenges. As a result, these fuzzers have a very high failure ratio at generating valid test cases to test the target IPC services. In other words, most IPC messages generated by the fuzzer fail to reach deeper code in target IPC services, not to mention to trigger potential vulnerabilities in them. We believe there are two challenges to address when fuzzing IPC services.

First, IPC services usually perform format checks on the incoming request messages, before actually processing these messages. For example, the handlers of MIG (Mach Interface Generator) Mach IPC messages in macOS have built-in inline code to verify the message headers. Without considering the format check, randomly generated IPC messages will quickly be rejected in the early stage of message handling. Therefore, the fuzzer has to generate well-structured IPC messages in order to find bugs in IPC services that may hide in deep.

Second, IPC services' message handling process is usually stateful, and a sequence of messages is required to trigger complicated actions. More importantly, the message sequence should be in a specific order (i.e., *ordering dependence*), and the parameters and return values of different messages are related to each other (i.e., *value dependence*). In order to explore the state machine of target IPC services and find potential vulnerabilities, the fuzzer has to generate IPC message sequences satisfying these dependences.

In this paper, we propose a new fuzzing solution to discover IPC bugs in IPC services without source code. To address the challenges above, we combine static and dynamic analysis to infer the knowledge of IPC messages, including message format constraints and message sequences' order and value dependences. With these knowledge, our fuzzer can better generate sequences of IPC messages to test target IPC services, and uncover deep bugs in IPC services.

More specifically, we use static analysis to analyze the binary code of target IPC services, in order to extract constraints of IPC message formats. One of the challenge to solve here is that, we have to first recognize the message handler functions, before extracting the IPC message format constraints. Our solution is following the message dispatching process. By feeding the extracted constraints to the follow-on message generators, we could use constraint solvers to generate valid IPC messages, in order to bypass the message format checks and reach deeper code during fuzzing.

Moreover, we use dynamic analysis to infer the dependences between IPC messages. First, we collect a large number of valid message sequences by monkey testing [7]. Then we infer the dependences between messages by building a probability matrix and a lookup table to model the transfer probability. Following this probability matrix and lookup table, our fuzzer could generate sequences of IPC messages satisfying the order and value dependences with a higher probability, and thus could trigger stateful actions in target IPC services as well as trigger hidden vulnerabilities.

To evaluate our design, we implemented a prototype named MACHFUZZER to fuzz a specific IPC service, i.e., WindowServer, a critical system component responsible for graphic operations in macOS. This IPC service processes complicated IPC messages, and is also a well-known target for attacks in real world. For example, it has been exploited multiple times in Pwn2Own contests in history, to escape the Safari's sandbox.

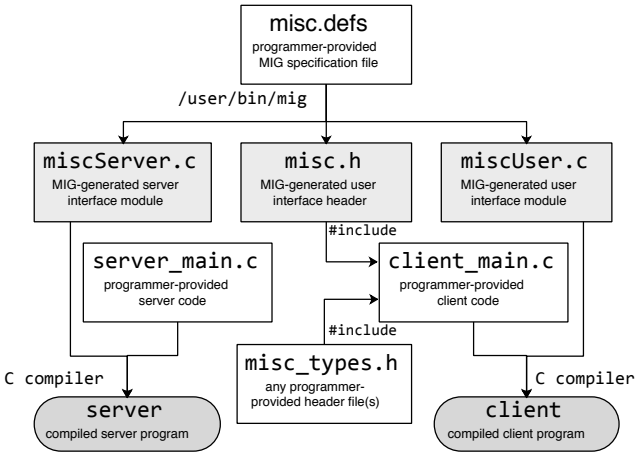

Fig. 2. Stub Generation Process of MIG IPC [8]

We evaluated our system on macOS High Sierra (version 10.13) in 4 virtual machines for 48 hours. The results showed that, MACHFUZZER is very effective at finding unknown vulnerabilities. It has discovered 12 unknown bugs on the latest macOS at the time of this writing. Among them, three vulnerabilities are confirmed exploitable, and could be exploited to escape the sandbox and gain root privilege.

Overall, this paper makes the following contributions:

1) We proposed a general fuzzing solution to test binary IPC services with knowledge inference.
2) We proposed a solution to use static binary analysis to extract IPC message format constraints and help construct valid IPC messages.
3) We proposed a solution to use dynamic analysis to infer constraints between IPC messages and help generate sequences of IPC messages.
4) We implemented a prototype MACHFUZZER and evaluated on an important IPC service WindowServer in macOS, and found 12 0day vulnerabilities.

II. BACKGROUND AND MOTIVATION

In this section, we first depict some fundamentals of IPC mechanisms. Also, we give a brief introduction to existing IPC fuzzers and explain why they cannot test IPC software effectively. We then summarize the challenges in IPC fuzzing.

A. Fundamentals of IPC

Interprocess communication (IPC) refers to the mechanisms provided by operating systems, which allows processes to manage shared data or collaborate. Typically, IPC involves two parties, i.e., a client and a server, where the client requests data or service and the server responds to client requests [9].

In general, IPC services are implemented in two different ways. First, commodity OSes and software provide interface

definition language (IDL) for developers to build IPC services, which is the most widely adopted way. Developers can specify message format and IPC interfaces in the IDL file, which will later be compiled to code stubs (including message format validation) for IPC client and server. A bunch of IDLs have been designed to fit different use cases, such as MIG [10], MIDL [11], and AIDL [12].

Let's take MIG (Mach Interface Generator) IPC, the IDL-based IPC in macOS, as an example. Fig. 2 shows the procedure of how MIG IPC is used to construct IPC client and server. First of all, developers define message type and prototype in IDL files (e.g. `misc.defs` in Fig. 2). Next, the IDL compiler `/usr/bin/mig` will compile the IDL files and generate code stubs for client and server (e.g. `misc-Server.c`, `miscUser.h` and `misc.h` in Fig. 2). Finally, developers fulfill the implementation based on the stub files.

The second way of implementing IPC service is reusing system APIs. Some OSes provide IPC programming framework for developers to build IPC services. Those IPC frameworks are usually encapsulation of primary IPC primitives, such as message sending and receiving. For example, XPC [13] is an IPC framework in macOS. Developers can register custom IPC services and define message handlers by calling XPC APIs.

The IPC message format is crucial for the communication. In macOS, an IPC message is consisted of a message header and a message body. The message header specifies the type of the message (i.e., ID), the size of message and some attributes (or flags) of the message.

B. IPC Fuzzing

Fuzzing, a highly popular software testing strategy, has become one of the de-facto standard strategies to uncover software vulnerabilities. In this section, we describe some existing IPC fuzzers and explain why they are ineffective to uncover deep bugs. In general, these fuzzers test the IPC server by sending a sequence of proper IPC messages that can be accepted by the target.

1) Hooking-based IPC Fuzzers: Some fuzzers attempt to fuzz the IPC services by intercepting IPC messages sending on the fly. We call such fuzzers hooking-based IPC fuzzers. Ret2 Systems implemented a fuzzer [6] that intercepts IPC messages delivered to WindowServer and injects bit flips into the intercepted messages. iSEC developed a fuzzing tool [5] that hooks primitives of named pipe IPC in Windows.

Fuzzers in this category generate test cases passively. Their code coverage primarily depends on the variety of intercepted IPC messages, which is hard to manipulate. Besides, even if a crash is found, it is hard to reproduce, since the original intercepted message is not recorded during fuzzing.

2) Grammar-based IPC Fuzzers: Existing fuzzers [2], [3], [14], [15], [16], [17] use pre-defined grammar template to generate structured messages. However, most of them need the assistant of source codes. Meanwhile, they fail to explore the dependence between messages. For example, the Chrome IPC fuzzer records some messages in the process and mutates the content by some pre-defined FuzzTraits templates. It could help the fuzzer to generate a lot of formatted messages blindly and correctly. But it fails to generate a sequence of dependence-aware messages dynamically. Chizpurfle [17] is designed for fuzzing proprietary Android services using a rich library of fuzz operators. It utilizes dynamic binary instrumentation to collect code block coverages as feedbacks, enabling the gray-box fuzzing. However, it did not consider the dependence between messages either.

```
1 // a1 is a pointer to a mach message
2 // Check if it is a complex message
3 if (*(_DWORD *)a1 >= 0
4     // check the number of descriptors
5     || *(_DWORD *)(a1 + 24) != 2
6     // check message size
7     || *(_DWORD *)(a1 + 4) != 76
8     // check the type of the first descriptor
9     || (v3 = -300, (*(_DWORD *)(a1 + 36) & 0
      xFF000000) != 0x1000000)
10    // check the type of the second descriptor
11    || (*(_DWORD *)(a1 + 52) & 0xFF000000) != 0
      x1000000
12    || ... ) { fail(); } // refuse the message}
13 else { acceptTheMessage(); }
```

Listing 1. Message Format Check Example of MIG IPC Handlers

C. Challenges of Fuzzing IPC Services

We encapsulate the challenges of fuzzing IPC services by studying some examples in macOS, and propose our general intuitions in designing MACHFUZZER to surmount these challenges. Without loss of generality, our insights are applicable to other IPCs as well.

1) Formulating Messages with Proper Format: Many IPC services validate the format of messages before handling them. Especially for IDL-based IPC, the format check code is added during IDL compilation. Without considering the message format, inputs generated by fuzzers will immediately be rejected. Some grammar-aware fuzzers have some basic ability to infer type information, but they rely on source code.

Listing. 1 presents an example of an MIG IPC handler, which is a pseudocode decompiled from the binary code by Hex-Rays decompiler. In the code snippet, it checks different fields of the mach message, such as message type, the message size and number of descriptors. If the message does not pass the check, the IPC invocation will fail.

The basic idea of our approach to tackle the problem is applying static binary analysis to extract the constraints of the message format, and use it to generate valid messages.

2) Identifying Message Handlers: To enable static analysis of message format constraints, we first need to identify message handlers. For open source IPC services, message handlers can be easily found by reviewing the source code. However, our design aims at fuzzing closed source IPC software. Given the IPC binaries with symbols stripped, collecting all the entries of IPC handlers will be challenging.

We discuss our basic idea to overcome the challenge according to how the IPC services are implemented. For IPC services

```
void makeScreenCapture() {
  CGDirectDisplayID displays[256];
  uint32_t dispCount = 0;
  CGImageRef img;
  CGDirectDisplayID dispId;
  // Invoke IPC handler _XGetDisplaySystemState()
  SLGetActiveDisplayList(256, displays, &dispCount);
  for (int i = 0; i < dispCount; i++) {
    dispId = displays[i];
    // Invoke IPC handler _XHWCaptureDesktop()
    img = SLDisplayCreateImage(dispId);
    ...
  }
}
```

Listing 2. Code snippet that utilizes the mach IPC to take screen captures

built on top of IPC framework, message handlers are registered and created via specific APIs. Handlers can be identified by tracking these API calls. For IDL-based IPC services, there is no explicit APIs to register handlers. Handlers can be identified by tracking the procedure of message dispatching.

3) Generating Dependence-aware Message Sequences: IPC message handling is stateful. A message may only be consumed when IPC service is in certain state. And to reach that state, other messages should be consumed first. In other words, to trigger some complicated IPC functions, a sequence of dependent messages are required. However, few IPC fuzzers have taken message dependence into account.

To understand more about dependence between IPC messages, we present an example in macOS. The code snippet in Listing 2 is utilizing IPC in macOS to take screen captures. The two functions prefixed with `SL` are client APIs provided by CoreGraphics library, which are used for sending IPC messages to corresponding handlers. The second IPC call `SLDisplayCreateImage()` takes in `dispId` as input, which is returned by the first IPC call `SLGetActiveDisplayList()`.

This example shows some operations like screen captures could be invoked only when we send IPC messages in a specific order with specific data, which highlights the difficulty in message generation. It's worth noting that, there are two types of dependences. The first one is *ordering dependence*, specifying in which order IPC messages should be sent. The second one is *value dependence*, specifying which value in the previous message reply should be used in the next message.

Ordering Dependence: From the previous example, the IPC handler `_XHWCaptureDesktop()` is invoked after the other handler `_XGetDisplaySystemState()`, as display ID should be retrieved first. We call such dependence as ordering dependence.

Value Dependence: Some IPC handlers take in data returned from another IPC handler. When a returned value of IPC handler A is used as input to a call to IPC handler B, we say that two IPC handlers have value dependence. From the previous example, display IDs returned by the handler `_XGetDisplaySystemState()` is used as input in the handler `_XHWCaptureDesktop()`.

Existing IPC fuzzers all focus on generating and mutating

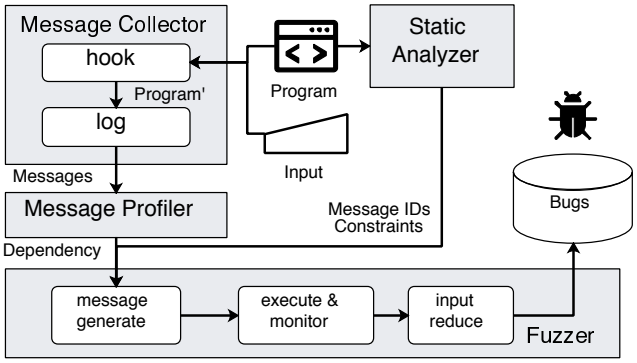

Fig. 3. Architecture of MACHFUZZER. The upper part is responsible for knowledge inference, while the lower part is the fuzzer based on the inferred knowledge. To infer the knowledge, static binary analysis is utilized to extract message format constraints, and dynamic analysis is utilized to collect sequences of messages and perform profiling.

a single message itself. To equip our fuzzer with dependence-aware message generation ability, the two types of dependence information should be inferred. The basic intuition to solve this challenge is to dynamically collect normal IPC messages and make analysis on them.

D. Techniques Involved

Monkey testing is a technique in software testing, where the user tests the application or system by providing random inputs (without knowledge or feedback) [7]. Monkey testing has been widely adopted in automated testing. In our scenario, we use it to trigger and capture IPC messages in the system.

Static analysis is a type of widely-used solutions which analyzes programs without actually executing them. It can be used to extract data-flow and control-flow information from programs [18]. Symbolic execution is another technique that interprets a program using symbolic values rather than concrete values. It is widely used in scenarios that require reasoning [19].

In this paper, we utilize all these program analysis techniques to learn the IPC message format, and construct valid messages to bypass sanity checks in programs under test.

III. SYSTEM DESIGN

In this section, we describe the holistic design of our IPC fuzzer MACHFUZZER.

A. Intuition

As aforementioned, to address the challenges of IPC fuzzing, we could use knowledge inference to guide fuzzing.

More specifically, we could use static analysis to extract the IPC message format constraints from binary code, then infer the dependence between IPC messages by analyzing a large number of valid message sequences with dynamic analysis.

Further, with the inferred knowledge of IPC messages, we could build a custom fuzzer to generate messages of valid format and generate sequences of dependence-aware messages to test target IPC services.

B. Architecture of MACHFUZZER

The overall architecture is shown in Fig. 3. MACHFUZZER is composed of 4 modules: an IPC message collector, an IPC message profiler, a static analyzer, and a fuzzer.

Message collector is used for logging IPC messages going through the target IPC service. The messages collected will be fed to message profiler as input. Message profiler is responsible for analyzing collected messages and generate dependences between messages and value constants in the messages. The first two modules establish a dynamic analysis procedure to assist the fuzzer in message generation.

The static analyzer first recognizes messages handlers and then analyzes the instructions in message handlers, generating constraints for different fields in messages. It can help the fuzzer to produce valid structured messages readily and effectively.

The message collector hooks target IPC services and records all incoming messages. Together with some techniques such as monkey testing [7], the collector could gather a large number of valid sequences of messages. Then the message profiler will analyze this corpus of message sequences to infer the dependences between messages.

Combining the results from the static and dynamic analysis, the fuzzer component is responsible for generating dependence-aware and format-valid IPC messages and sending them to the target IPC services. For the convenience of bug analysis, an input reducer is also integrated here to help extract simplified proof of concept (PoC) messages.

In the following subsections, we describe each module in detail, including design intuitions and decisions.

C. IPC Message Collector

As aforementioned, hooking-based fuzzers usually hook the target IPC services, to collect valid IPC messages for further mutation. These fuzzers usually mutate the collected messages randomly and inject the forged messages into the hijacked communication channel.

In our fuzzer, we can also take advantage of this hooking-based technique to collect IPC messages for further use in dependence analysis. We have addressed the following two key points in designing the message collector.

First, we need to find a hooking point to log the full content of all incoming and outgoing messages at runtime. Second, we need to generate a large number of valid message sequences. It is obvious that, the bigger amount and variety of messages we collect, the better result of dependence analysis will be.

Hence, to produce IPC messages as many as possible, and as diversified as possible, we should drive various IPC clients to send various IPC requests to the target IPC server, by triggering different IPC operations in the system. In other words, we have to make the target IPC service busy.

1) IPC Message Logging: For both framework-based and IDL-based IPC, there are unified interfaces for clients to send and receive messages. All IPC messages go through these interfaces, which are IPC primitives implemented in the form of library APIs or system calls. For example, MIG IPC relies on the system call `mach_msg` in macOS to send and receive messages.

Therefore, to monitor and record all the IPC messages passed in and out of the target IPC server, we hook these IPC APIs or system calls. In the hooks, we also follow fields in the message object that are potentially pointers, and record the pointees. For the purpose of value dependence analysis, we also record message responses.

2) Input Generation: Different IPC services respond to different input operations. Lots of IPC services are crucial system services that respond to hardware related events, such as GUI events, USB events, disk IO events, Wi-Fi events, etc. We adopt a monkey testing based methodology to trigger the events, i.e, randomly generate input to simulate related events.

Let's take WindowServer in macOS, as an example to show how we generate inputs. WindowServer is responsible for handling graphics operations, listening to requests via Mach IPC. Any GUI event will be routed by the system to send IPC messages to WindowServer. GUI events include window transformation, cursor movement, desktop switching etc., which can be triggered by mouse and keyboard inputs. Thus we simulate mouse and keyboard events as a *monkey*, and wrap the primitive events to implement high-level GUI operations like window opening, resizing, minimizing, maximizing and dragging. By randomly and repeatedly produce mouse and keyboard inputs and high-level primitives, we could get a bunch of valid IPC messages generated and sent to the IPC service, which will then log the messages.

D. IPC Message Profiler

IPC message profiler aims at figuring out the ordering and value dependences between IPC messages, by learning from the collected messages. Furthermore, integer value and string constants are also extracted and put into a dictionary for further message generation.

1) Ordering Dependence Analysis: The ordering of IPC messages matters, because some actions need to be fulfilled by sending multiple IPC messages in specific order. For example, thinking about a usage scenario in WindowServer, before setting the property of a window by invoking the IPC handler `_XSetWindowProperty`, the user/client needs to retrieve the window ID by invoking `_XFindWindow` or `_XCreateWindow` first. If the sending order between the two messages changes, the IPC service handlers will reject one of the messages.

We quantitatively analyze the ordering dependences between each pair of messages, with a novel message ordering dependence probability matrix, as shown below.

$$ProbabilityMatrix: \begin{bmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,N} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ P_{N,1} & P_{N,2} & \cdots & P_{N,N} \end{bmatrix}$$

In the matrix, each row and column corresponds to a message ID (i.e., type). Let ID_i be the message ID corresponding to the ith row and ith column. Suppose we have a total number of N messages with different IDs, the matrix should be N x N. In the matrix, the number located at ith row and jth column is annotated as $P_{i,j}$, which stands for the probability that the message ID_j will appear after the message ID_i.

Assuming two messages have ordering dependence, it means one message should be sent after another. With the long sequence of collected messages, we use a sliding window to go through the message queue. The size of the sliding window is determined according to the complexity of the state transitions in IPC services. In our evaluation, the target macOS IPC services are not too complicated, and messages that have ordering dependence are usually close to each other. So we choose a small size of 10 for the sliding window in our implementation. In each window, if message ID_j is after message ID_i, we increase $P_{i,j}$ by 1. In the end, we could calculate the $P_{i,j}$ value for each ordered pair of message IDs. In the last step, $P_{i,j}$ in the matrix should be normalized to make the sum of the value in each row (or column) equals to 1.

Message ordering dependence probability matrix roughly estimats how likely a message follows another, which can direct the fuzzer to simulate stateful actions.

2) Value Dependence Analysis: Return values (including those returned in function arguments) of a precedent message, could be the input parameters of another following message. This is called value dependence. We employ some heuristics to analyze value dependences between each ordered pair of messages.

In each sliding window of the collected message sequence, after counting the ordering dependence probability, we can make an additional step to infer the value dependence. More specifically, we will check if the return values of the first message and the contents of the second message share the same integer value or string. If it is true, we will record the two locations for the matched value.

Thus we can build a lookup table to store these value dependences. In the table, each row stands for a pair of ordering dependent messages, and the potential value dependence between this pair will be recorded in this row. Later, when generating IPC messages, in addition to the ordering dependence, we would put proper context-sensitive values into the messages following the lookup table.

In our current prototype, only integer value and null-terminated ASCII strings are supported. We leave value dependence analysis on more advanced data structures into future work.

3) Constants Extraction: Messages may contain constants that are shared across all the messages with same ID (i.e., type). The constant or enumeration values may be hard for the static analyzer to find out from the binary. However, from the message logs collected by dynamic analysis, we can go through the messages with same ID and easily extract the common value as constants to the message.

With these constants, we could further generate valid messages with correct constants values in specific fields.

E. Static Analyzer

Static analyzer first identifies all the message handlers in the binary of the IPC server, then it extracts a list of message IDs with constraints by symbolic execution.

1) Identifying message handlers: As mentioned above, framework-based IPC uses specific APIs to register and create message handlers. Typically, handler functions and message IDs are parameters of these API function. By statically tracking these API invocations, we could find all the message handlers registered in the IPC server.

To identify message handlers for IDL-based IPC, static analyzer first locates the message dispatcher in the binary of the IPC server, by statically tracking the invocations of the handler that is responsible for incoming IPC processing. This IPC handler is typically a dispatcher, which uses a dispatch table to dispatch different message IDs to different handlers. Our static analyzer then leverages a heuristic method to search for a pointer referenced by the dispatcher, which meanwhile points to an array of pointers. This array is likely to be the dispatch table. Finally, by traversing the dispatch table, a list of message IDs and handlers can be recognized.

2) Extracting constraints: Code for message format check usually appears at the very beginning of a message handler. After finding the message handlers, we can quickly locate code entries for message format check. Typically, the verification code contains a number of conditional statements to check common fields in the message header, such as message type and message size, just as the example illustrated in Listing 1.

The goal for static analyzer is to find code paths that can make the message pass the format verification. Symbolic execution can be easily applied in this scenario. Our static analyzer first symbolizes the input message, then explores the execution space by symbolic execution. Once the code paths exploration reaches the points where format verification is passed, the static analyzer records the constraints along the execution path.

F. Fuzzer

The fuzzer module takes in ordering and value dependences (i.e. ordering dependence probability matrix and value dependence table), and a list of supported message IDs with constraints for each message ID as input. In the following subsections, we will describe how we generate IPC messages for testing, and how we reduce the input for bug analysis when a crash is identified.

1) Test Message Generation: For each test, we will send a maximum of T messages (T is set to 10000 in our experiment). For each message, we generate the full content by following the steps below:

(I) In the first step, we select the message ID to send. We have two modes to decide which message ID to choose, random mode and dependence-aware mode. In random

mode, we simply choose a message ID from the full list randomly. In dependence-aware mode, we could choose the message ID according to the probability value in the message ordering dependence probability matrix. Suppose we send a message of ID X in the previous round, we can look up the ordering dependence probability values for each message ID sent after message ID X, and choose one of them in the probability accordingly. We could switch between the two modes randomly in order to achieve good message variety and ordering dependence at the same time. In our implementation, we use the two modes in probabilities of half and half.

(II) In the second step, we use constraint solver to solve the constraints of selected message ID generated by static analyzer, and filled the results in corresponding fields of the message. For example, the fields of message type and size for selected message ID will be figured out from the constraints in this step.

(III) When the message ID is selected in dependence-aware mode, we may also have value dependences between the previous message and current message. By looking up the value dependence table, we could infer which integer or string value in the reply of previous message is used in current message. This dependent value also can be slightly mutated (e.g. bit flipping). When the message ID is selected in random mode, we do not generate dependent values in the message.

(IV) Constants extracted by message collector and message profiler could be used to fill into the corresponding location in the message.

(V) In the last step, we still have unspecified data in the message, which is not detected to have constraints in the format check, value dependences and constant values. We just fill them with randomly generated data.

We logged the full content of every message we sent. When a crash is triggered, proof of concept input could be reproduced for bug analysis.

2) Input Reducing: In our crash logs, the input usually contains thousands of messages. It is extremely difficult to figure out the root cause of the crash, since too much noise input is involved.

To remove the irrelevant messages in the proof of concept and effectively reduce the input, we employ a simple but effective fix-point algorithm—bisection, which has been wieldy utilized in the community. When start reducing, the reducer will select 1/2 in a test case randomly. If the rest part can trigger the bug continuously, the reducer will reject the selected content then starts a new iteration. By contrast, if it cannot trigger the bug again, the reducer will select 1/4, 1/8, 1/16, ... , 1/n of the whole content randomly until it can trigger the bug again. As a result, we could fetch a minuscule reproducible case which can trigger the bug steadily.

IV. Implementation

MachFuzzer is implemented in a mixture of Python and Objective-C and composed of several compartmentalized building blocks.

a) Message collector and profiler.: The messages are traced and collected by a script of Frida DBI framework. For input generation, we use PyUserInput library to trigger keyboard and mouse events and manipulate graphics layout automatically. The message profiler is written in Python.

b) Static analyzer.: To automatically identify and locate the message handlers, we use IDA Python APIs to extract some necessary information of it. Besides, we utilize IDA hex-rays to obtain the decompiled code of entries of message handlers; and thus, we use KLEE, a symbolic execution engine based on LLVM, to generate numerous proper message headers.

c) Fuzzer and infrastructures.: We run the fuzzer inside macOS virtual machines. To avoid an explosion of recorded messages during fuzzing, we specify an upper limit on the number of messages sent in each round of fuzzing. Before each round of fuzzing, we reset the fuzzing environment to a clean state by reverting to a prepared virtual machine snapshot, which is automated by VMware API. Moreover, the component responsible for generating and sending message sequences is implemented in Objective-C, while the status of target IPC servers is monitored by LLDB-Python scripts.

V. Evaluation

We performed empirical evaluations to answer the following questions regarding the efficiency of MachFuzzer.

(I) Can MachFuzzer find realistic vulnerabilities?

(II) Is dependence analysis useful in finding the vulnerabilities?

(III) Compared to existing mach IPC fuzzers, how does MachFuzzer perform?

A. Experimental Setup

To evaluate the bug hunting capability of MachFuzzer, we ran it on 4 virtual machines running macOS High Sierra version 10.13 and 4 virtual machines running macOS Mojave 10.14, on 4 vCPUs with 8 GB of memory. The host machine is running VMware ESXi 6.5 on dual Intel Xeon E5-2683v3 CPUs with 128 GB of memory. Meanwhile, the fuzzer has been applied to WindowServer, a closed source system component in macOS.

1) Dynamic Analysis (Profiler): We ran the message collector for half an hour in a single virtual machine and collected around 500,000 messages triggered by generated input. It took message profiler about 10 minutes to generate the results, including message ordering dependence probability matrix, value dependence table and possible constants for each message ID. We choose sliding window size as 10.

2) Static Analysis: We ran the static analyzer on the binaries, which contains all the code of IPC handlers. It took around 10 minutes to extract the constraints for a total of 584 message handlers.

Idx	Crash Point	IPC Handler	Error Type	Impact
1	_XHWCaptureWindowListToIOSurface	_XHWCaptureWindowListToIOSurface	NULL pointer deref	DoS(NULL)
2	WSWindowHasActiveBlur	_XHWCaptureWindowListToIOSurface	NULL pointer deref	DoS(NULL)
3	PKGSpaceIsOrderedIn	/	EXC_BAD_ACCESS	DoS
4	CFHash	_XGetWindowProperty	NULL pointer deref	DoS(NULL)
5	__PKGSpacesWindowDidUpdateConstraints_block_invoke	_XPackagesSetWindowConstraints	Assert error	DoS
6	x_list_create_cf_type_array	_XCopyManagedDisplaySpaces	EXC_BAD_ACCESS	DoS
7	PKGWindowSetMovementParent	_XSetWindowOriginRelativeToWindow	NULL pointer deref	DoS(NULL)
8	_PKGSpaceContainsWindow_block_invoke	_XPackagesAddWindowToDraggingSpace	Use after free	exploitable
9	PKGSpaceIsOrderedIn	_XSetWorkspace	Type confusion	exploitable
10	__CFTypeCollectionRelease	_XSetConnectionProperty	Out of bound access	exploitable
11	__CFBasicHashAddValue	_XRegisterClient	Integer Overflow	DoS
12	_XGetDebugOption	_XGetDebugOption	Out of bound access	infoleak

TABLE I
UNIQUE WINDOWSERVER BUGS FOUND ON MACOS WITH MACHFUZZER

3) Fuzzing: After static and dynamic analysis results were ready, we ran the fuzzer on all the 8 virtual machines for 48 hours. For each time of fuzzing, we generate a maximum of 10000 messages. Hundreds of crashes were produced.

B. Bugs Discovered

We categorized the crashes according to the context information of the crash scene and do manual investigation. Finally, we figured out MACHFUZZER has found a total number of 12 unique bugs that are unknown previously. Combining manual analysis with crash log, the bugs have a variety of error types including NULL pointer dereference, bad access of memory, assert error, use after free, integer overflow, and type confusion, as summarized in Table I.

The second column shows which function the crashing instruction belongs to. We can see in the table, the memory error could happen in different functions that belong to different libraries, since WindowServer replies on multiple libraries.

The third column indicates which IPC handler each crash happens in. The bug of index 3 crashes in a callback function instead of any IPC handler.

The fourth column describes the cause of each crash. Most of the crashes were due to memory access error except one is caused by assert error. By manually analyzing the memory errors, we filled more specific vulnerability types in the table. Other than the simple NULL pointer dereference bugs, MACHFUZZER has found 4 more advanced memory bug types: use after free, type confusion, out of bound access and integer overflow.

Finally, the fifth column depicts the impact of each bug. We manually investigated each of them, and label them either with "DoS" or "exploitable". For the bug labeled with "infoleak", we have written a working exploit to achieve address information leakage. For each of the 3 bugs labeled with "exploitable", we have written some working exploits to achieve sandbox escape and privilege escalation. It is worth mentioning that, we have successfully chained one of the 3 bugs with Safari exploit to demo a full remote root attack against macOS in Pwn2Own 2017. All the bugs have been reported to Apple and got fixed. The last 5 bugs have been assigned CVE numbers CVE-2017-2537, CVE-2018-4449, CVE-2018-4450, CVE-2018-4415, and CVE-2019-6220, respectively. Note that CVE-2018-4415 (Integer Overflow in `CA::Render::InterpolatediFunction::InterpolatedFunction`) was not assigned to us, because another security researcher reported it before us.

```
void trigger() {
  // Invoke IPC handler
  // _XPackagesAddWindowToDraggingSpace()
  SLSPackagesAddWindowToDraggingSpace(window_id1);
  // Create a new space
  // Invoke IPC handler _XSpaceCreate()
  int sid = SLSSpaceCreate(gDefaultCID, 0, 0);
  // Destroying Space
  // Invoke IPC handler _XSpaceDestroy()
  // Free the Dragging Space
  // leaving a dangling pointer
  SLSSpaceDestroy(gDefaultCID, sid - 1);
  // Invoke IPC handler
  // _XPackagesAddWindowToDraggingSpace()
  // Use the dangling pointer
  SLSPackagesAddWindowToDraggingSpace(window_id2);
}
```

Listing 3. Proof-Of-Concept of CVE-2017-2537 that was discovered by MACHFUZZER

C. Case Study

To understand how dependence analysis helps us to find bugs, let's take one of the 3 exploitable bugs as an example.

The use after free bug CVE-2017-2537 in WindowServer can be triggered by 4 IPC messages. In the code snippet from Listing 3, all the functions that are prefixed with `SLS` are client APIs, which are equivalent to sending an IPC message to the corresponding handler. For example, calling `SLSSpaceCreate` is equivalent to sending an IPC message to `_XSpaceCreate`. For convenience, we do not show the full content of the messages that can trigger the bug, we use client API calls to show the specific steps to trigger the bug.

The first IPC message, i.e. a call to SLSPackagesAddWindowToDraggingSpace() allocated a structure on the heap, which is dragging space. Then a new space is created by sending the second IPC message, i.e. calling `SLSSpaceCreate()`. Space ID (`sid`) is returned. The third IPC call invokes `SLSSpaceDestroy()` with parameter `sid - 1`, which refers to the dragging space allocated in the first step so that the dragging space is deallocated, leaving a dangling pointer. Finally, the last IPC message triggers the use of freed dragging space and lead to crash.

In the process of producing the use after free vulnerability, the IPC messages have both ordering and value dependences. IPC handler `_XSpaceDestroy()` should be invoked after `_XPackagesAddWindowToDraggingSpace()` and `_XSpaceCreate()`. Moreover, the integer space ID number passed to the handler `_XSpaceDestroy()` in the IPC message, is dependent on the return value of `SLSSpaceCreate()`, which is also a space ID number returned in the IPC handler `_XSpaceCreate()`'s reply.

By inferring the ordering dependences and value dependences, we can help the fuzzer to generate complicated input that can trigger deep bugs. The use after bug above is a good example for dependence analysis.

D. Comparison against Existing Fuzzer

How does MACHFUZZER compare with existing IPC fuzzers? To answer this question, we compared MACHFUZZER against the fuzzer described in the blog of Ret2 Systems [6], which is the only existing IPC fuzzer in macOS prior to our work. Although Ret2 Systems's fuzzer is not open sourced, their idea is simple enough to reimplement the tool. Another reason is that it also targets WindowServer and has already found one exploitable bug.

Ret2 Systems's fuzzer is a hooking-based fuzzer. By installing API hooks in WindowServer, it mutates IPC messages by bit flipping. The fuzzer does not generate the message from scratch, and all the input comes from IPC clients running on the system. To increase the variety of message traffic to WindowServer, they simply hold the 'Enter' key to trigger GUI events.

We reimplemented Ret2 Systems's fuzzer according to the blog and ran it on the same machines for the same time duration. It discovers only 2 Null pointers dereference bugs, which also can be found by MACHFUZZER. MACHFUZZER can find 5 times more bugs than this fuzzer.

After further investigation, we found that it is reasonable that Ret2 Systems's fuzzer did not do well. Hooking-based fuzzer does not generate message itself, so the variety of message types for fuzzing is limited. Holding the 'Enter' key, we counted the number of different message IDs received by WindowServer during fuzzing. The resulting number is 92/584 (15.76%). The message collector of MACHFUZZER is also running in the similar way with hooking-based fuzzer, but by running an input generation module, we have collected 292/584 (51.03%) messages, which is 3 times more than Ret2 Systems's fuzzer. It proves that our input generation module effectively increased the amount and variety of collected IPC messages.

VI. RELATED WORK

A. Mutation-Based Fuzzing

Mutation-based fuzzing approaches such as AFL-family are proposed to produce reasonable test cases by mutating some well-formed input cases. AFL, AFLgo, and OSSFuzz [20], [21] use code coverage to guide the mutation of corpus. CollAFL [22] improves the bitmap algorithm to reduce the edge collision ratio to nearly zero. AFLFast [23] utilize markov chain to improve code coverage. Angora [24] uses gradient descent algorithm to solve constraints and improve code coverage. Driller [25] uses symbolic execution when fuzzing is stuck. They could generate high code coverage with the guidance of feedbacks. However, most of them need the assistant of source codes to apply necessary instrumentation. Besides, they cannot generate dependence-aware workloads. MACHFUZZER utilizes the basic concept of it to examine IPC servers with proper message headers formulated by symbolic execution.

B. Generation-Based Fuzzing

Generation-based fuzzing is effective to discover vulnerabilities in interpreters, compilers and parsers. Numerous studies [26], [27], [28] leverage pre-defined templates and rules to generate html, css, or javascript codes to fuzz rendering engine of browsers. They could generate numerous correct input readily. However, these fuzzers do not take the dependence of the execution context into account.

C. IPC Fuzzing

Fuzzing has been effectively applied in IPC fuzzing for years, and lots of bugs in various IPC implementations have been discovered. According to the different techniques employed, previous works in IPC fuzzing can be categorized into the following classes.

1) Hooking-based IPC Fuzzers: Hooking-based IPC fuzzers [5], [6] do not construct messages from scratch. They capture and mutate test cases in a passive way.

Their advantage is that mutations all happen on valid messages, and inputs are not rejected quickly. However, compared with generation-based fuzzing and our technique, hooking-based methodology has lower message type coverage, since it replies on the variety of messages that can be captured. Moreover, it also has a drawback in reproducing the input when a crash has been found.

2) Grammer-based IPC Fuzzers: Grammer-based IPC fuzzers [2], [3], [15], [16] utilize grammar or structure information to facilitate IPC message generation. However, the grammar template must be written by human researchers with knowledge from manual analysis of source code. All the previous IPC fuzzing tools in this category reply on source code, IDL decompiler or manual analysis to infer the message structure, and fail to consider message dependences, while our fuzzer performs knowledge inference directly from binaries.

D. Static Analysis of IPC Services

There are also several static analysis approaches dedicating to find logical issues in IPC services. Kratos [29], AceDroid [30] uses static analysis frameworks to examine the permission issues in Android services. [31] utilizes differential analysis algorithms to detect security configuration changes introduced by Android customization. Invetter [32] leverages static analysis to validate the secure use of sensitive input validations in Android framework.

VII. Discussion

The ideas we have proposed in this paper for fuzzing IPC is general and can be applied to other systems. However, there exits some factors that will affect the implementation. First, static analyzer and symbolic engine should be altered to adapt to the programming language used in targeting IPC. Second, since different IPC services respond to different input operations, input generation in monkey testing should fit the functionalities of targeting IPC services. Last, the basic executor and monitor of the fuzzer should be modified to run under the targeting IPC mechanism and operating systems.

VIII. Conclusion

We present a novel IPC fuzzer that can infer format of messages and dependence information between messages to improve the effectiveness of bug discovery. By performing static analysis on code of message format checks, our fuzzer constructs IPC messages of valid formats. By performing dynamic analysis on messages captured during monkey testing, our fuzzer extracts dependence information and help to generate stateful IPC messages. Then we implemented MachFuzzer, and evaluated it on macOS. As a result, we found 12 previously unknown vulnerabilities in WindowServer. Our experiments show that our method has a practical impact on finding IPC vulnerabilities.

IX. Acknowledgment

We would like to thank our shepherd, Domenico Cotroneo, and the anonymous reviewers for their comments and advice. This work was supported in part by National Natural Science Foundation of China under Grant 61772308 and U1736209, BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009, and Tsinghua University Initiative Scientific Research Program under Grant 20151080436.

References

[1] iOS/macOS Safari Sandbox Escape via QuartzCore Heap Overflow. https://blogs.securiteam.com/index.php/archives/3796. Last accessed Feb. 2019.

[2] G. Gong. Fuzzing android system services by binder call to escalate privilege. https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf. Last accessed Feb. 2019.

[3] Q. He. Hey your parcel looks bad fuzzing and exploiting parcelization vulnerabilities in android. https://www.blackhat.com/docs/asia-16/materials/asia-16-He-Hey-Your-Parcel-Looks-Bad-Fuzzing-And-Exploiting-Parcelization-Vulnerabilities-In-Android.pdf. Last accessed Feb. 2019.

[4] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "Intentfuzzer: detecting capability leaks of android applications," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 531–536.

[5] J. Burns. Fuzzing Win32 Interprocess Communication Mechanisms. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Burns.pdf. Last accessed Feb. 2019.

[6] B. Patrick and G. Markus. (2018) Cracking the Walls of the Safari Sandbox. https://blog.ret2.io/2018/07/25/pwn2own-2018-safari-sandbox/. Last accessed Feb. 2019.

[7] B. Hofer, B. Peischl, and F. Wotawa, "Gui savvy end-to-end testing with smart monkeys," in *2009 ICSE Workshop on Automation of Software Test*. IEEE, 2009, pp. 130–137.

[8] J. Levin, *Mac OS X and iOS Internals: To the Apple's Core*, 1st ed. Birmingham, UK, UK: Wrox Press Ltd., 2012.

[9] Microsoft. Interprocess Communications. https://docs.microsoft.com/en-us/windows/desktop/ipc/interprocess-communications. Last accessed Feb. 2019.

[10] M. R. T. Richard P. Draves, Michael B. Jones. MIG - The MACH Interface Generator. http://www.cs.cmu.edu/afs/cs/project/mach/public/doc/unpublished/mig.ps. Last accessed Feb. 2019.

[11] Microsoft. Microsoft Interface Definition Language. https://docs.microsoft.com/en-us/windows/desktop/Midl/midl-start-page. Last accessed Feb. 2019.

[12] A. D. Documentation. Android Interface Definition Language (AIDL). https://developer.android.com/guide/components/aidl. Last accessed Feb. 2019.

[13] Apple. Apple XPC. https://developer.apple.com/documentation/foundation/xpc. Last accessed Feb. 2019.

[14] S. E. Lab. RPC Forge. https://github.com/sogeti-esec-lab/RPCForge. Last accessed Feb. 2019.

[15] Google. Chromium IPC fuzzer. https://chromium.googlesource.com/chromium/src.git/+/65.0.3283.0/docs/ipc_fuzzer.md. Last accessed Feb. 2019.

[16] N. williamson. Chromium AppCache Fuzzer. https://cs.chromium.org/chromium/src/content/browser/appcache/appcache_fuzzer.cc. Last accessed Feb. 2019.

[17] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 1–11.

[18] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Systems Journal*, vol. 46, no. 2, pp. 265–288, 2007.

[19] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[20] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[21] Google. OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Last accessed Feb. 2019.

[22] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[24] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[25] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 01 2016.

[26] Mozilla. Dharma: A generation-based, context-free grammar fuzzer. https://github.com/MozillaSecurity/dharma. Last accessed Feb. 2019.

[27] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," vol. 9878, 09 2016, pp. 581–601.

[28] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 445–458.

[29] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework." in *NDSS*, 2016.

[30] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "Acedroid: Normalizing diverse android access control checks for inconsistency detection." in *NDSS*, 2018.

[31] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android roms via differential analysis," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 1153–1168.

[32] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1165–1178.