IMF: Inferred Model-based Fuzzer

HyungSeok Han
KAIST
hyungseok.han@kaist.ac.kr

Sang Kil Cha
KAIST
sangkilc@kaist.ac.kr

ABSTRACT

Kernel vulnerabilities are critical in security because they naturally allow attackers to gain unprivileged root access. Although there has been much research on finding kernel vulnerabilities from source code, there are relatively few research on kernel fuzzing, which is a practical bug finding technique that does not require any source code. Existing kernel fuzzing techniques involve feeding in random input values to kernel API functions. However, such a simple approach does not reveal latent bugs deep in the kernel code, because many API functions are dependent on each other, and they can quickly reject arbitrary parameter values based on their calling context. In this paper, we propose a novel fuzzing technique for commodity OS kernels that leverages inferred dependence model between API function calls to discover deep kernel bugs. We implement our technique on a fuzzing system, called IMF. IMF has already found 32 previously unknown kernel vulnerabilities on the latest macOS version 10.12.3 (16D32) at the time of this writing.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*;

KEYWORDS

fuzzing; kernel vulnerabilities; model-based fuzzing; API fuzzing

1 INTRODUCTION

Kernel vulnerabilities are recently gaining significant levels of attention. According to the National Vulnerability Database, there were 248 kernel vulnerabilities reported in 2015, but the number doubled in 2016 to 478. Furthermore, the attack surface of modern OS kernels is expanding as their codebase sizes rapidly increase [32]. For instance, a recent report [14] indicates that the Linux kernel is accepting 4K lines of new code every day. Recent advances in kernel exploitation [23, 28, 58] have also been highlighting the popularity of kernel vulnerabilities.

Although there has been much research on discovering kernel vulnerabilities, most of the advances involve source code auditing, and therefore, not applicable to commodity OSes such as Windows or macOS. KLEE [7] symbolically executes the source code to find vulnerabilities, and it has been used to check the HiStar [61] kernel. There are static analyzers such as CQUAL [24] and KINT [54], the aim of which is to find kernel vulnerabilities. There are also kernel-level formal verification approaches [29, 30], but all such techniques require the source code.

The current best practice for finding kernel vulnerabilities on commodity OSes is fuzzing, which involves repeatedly calling an arbitrary sequence of kernel API functions, e.g., system calls, with randomly generated parameter values. There are various kernel fuzzers developed by security experts and practitioners [4, 25, 37, 40, 53, 57] as well as academic researchers [17, 19, 55], but all of them share the same idea: they call random kernel API functions with randomly generated parameter values.

Unfortunately, all of the existing kernel fuzzers are associated with a high failure rate in terms of how many API functions can be executed without an error. API function calls fail when given an invalid calling context. For example, a call to `write` will always fail if there is no prior call to `open`. Furthermore, an `open` call should return a valid file descriptor with the write permission, and a `write` call should take in the returned file descriptor as the first argument in order to be successful. Without considering such cases, it is unlikely that one will find bugs that are latent deep in the kernel.

The same problem arises in userland fuzzing. Suppose we are fuzzing a program that takes in a PNG file as input. One can feed in randomly generated files to the program, but the program will quickly reject the files because they do not follow the specifications of the PNG format. Therefore, we need to provide randomly mutated yet well-structured inputs in order to find deep bugs in the program under test. We can easily draw an analogy between a sequence of kernel API calls and a sequence of input fields of an input file. The key challenge with regard to kernel fuzzing is that (1) the ordering of kernel API calls should appear to be valid, and (2) the parameters of API calls should have random yet well-formed values that follow the API specification.

To address this challenge, we propose a model-based approach to kernel fuzzing. The idea of model-based fuzzing is not new. Many userland fuzzers [2, 16, 20, 27, 62] indeed use an input model for generating test cases. However, automatically generating such a model for fuzzing is an active research area [12, 18], and has never been applied to kernel fuzzing. This paper presents the first model-based kernel fuzzer that leverages an automatic model inference technique. Specifically, we analyze kernel API call sequences obtained from running a regular program to deduce an API model. We then use the model to generate test cases repeatedly in a fuzzing campaign.

Our approach is inspired by the dynamic nature of kernel API calls, in which kernel API logs taken from the executions of the same program with the same input can vary. The call sequences can diverge due to thread interleaving. The parameter values can change depending on the parameter type or due to non-deterministic OS-level features such as Address Space Layout Randomization (ASLR). For example, with regard to running `strace` on a program with the

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134103

same input twice, we can have two distinct system call traces from two executions. However, such non-determinism introduces another opportunity for us to infer constant factors from API function calls, e.g., we can infer which parameter value is a constant. We leverage this intuition so as to automatically infer an API model.

In this paper, we present the design and implementation of IMF, the first *model-based* fuzzer that runs on macOS. Although our current implementation is specific to macOS, the proposed technique is general and can be applied to other operating systems. We evaluated our system on macOS Sierra version 10.12.3 (16D32), which was the latest version at the time of this writing, with API logs obtained from hundreds of applications downloaded from the App Store. We ran fuzzing for 1,740 hours on 10 Mac minis. As a result, IMF found a total of 32 previously unknown kernel panics on macOS Sierra. Our experiment also shows that IMF can find 3× more bugs than an existing kernel fuzzer [4] in the same amount of time.

Overall, this paper makes the following contributions:

(1) We introduce a novel method called *model-based API fuzzing*, which exploit the similarity between API logs to produce random yet well-structured API call sequences.
(2) We implement our idea in a prototype called IMF, which is the first model-based API fuzzer for testing commodity OS kernels.
(3) We evaluate IMF on macOS, and found 32 kernel vulnerabilities that lead to a kernel panic.
(4) We make our data and source code public in support of open science: https://github.com/SoftSec-KAIST/IMF.

2 BACKGROUND

In this section, we start by reviewing the concept of API fuzzing. We then summarize the history of kernel fuzzing, and the motivation behind our research. Lastly, we briefly describe the internal structure of macOS, which is the main target of this paper.

2.1 API Fuzzing

API fuzzing has garnered less attention compared to traditional fuzzing because it does not guarantee sound results. Traditional fuzzers are sound; when they find a bug, it is truly a bug. On the other hand, API fuzzers are not sound, because the bugs found can be false. For example, given a buggy function f that takes in an integer value as input and crashes when the value is greater than 100, if a program exists that always calls f with a constant integer 0, we cannot say that there is a bug in the program despite the fact that the program calls the buggy function f, because the bug will never trigger in the program.

The key problem is that user-level API functions are not on the attack surface. Although an attacker can write an arbitrary program that calls the buggy function f with a buggy input, the attacker cannot run the program on a victim's machine. If the attacker has the ability to do so, then the attacker had already gained the privileges of the victim.

However, things change when it comes to kernel fuzzing, as kernel API calls allow an attacker to cross a trust boundary [41] between the user and the kernel space. In particular, an attacker can exploit kernel API bugs to gain unprivileged root access. Local privilege escalation can be useful after a successful remote exploitation. For example, attackers who obtain remote user-level access, can further exploit kernel vulnerabilities in order to install a kernel rootkit on the victim's machine.

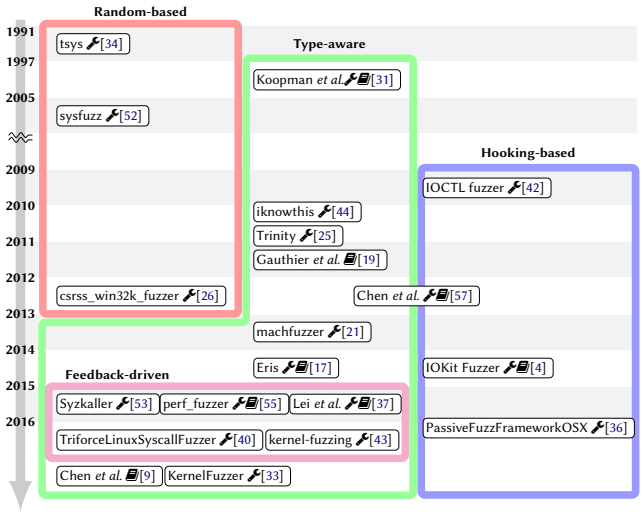


Figure 1: The history of kernel fuzzing. 🔧 denotes that a tool was created. 📄 denotes that a paper describing the work was published.

2.2 Kernel Fuzzing

Despite its popularity, there have been few systematic studies of kernel fuzzing. Many kernel fuzzers provide their source code, but the lack of a concise description of their algorithm and clarification of their design decisions makes it difficult for other developers to learn from existing works. As a result, similar methodologies are repeatedly appearing even up to present.

In this paper, we summarize the history of kernel fuzzing and highlight recent advances in this area. Figure 1 presents the overall history of kernel fuzzing. We define four major categories: random-based, type-aware, hooking-based, and feedback-driven fuzzing.

Random-based Kernel Fuzzers. The history of kernel fuzzing dates back to 1991, when Tin Le released the tsys fuzzer [34], which was simply written in about 250 lines of C code. The idea of the fuzzer is simple: it invokes a series of randomly selected system calls to UNIX System V with randomly generated arguments. The same design even appears nowadays: [26, 52] also use the same methodology. We refer to fuzzers in this category as *random-based kernel fuzzers*.

Type-aware Kernel Fuzzers. Koopman *et al.* [31], in 1997, compared the robustness of OSes with a finite set of manually chosen test cases for system calls. The test cases were carefully generated based on their type. For example, they used seven predefined values for types of file handles including an opened handle for read operations, an opened handle for both read and write operations, and a closed handle. Similarly, each parameter type, such as buffer,

length, and file mode, had a set of designated values. They tested each system call with all possible combination of predefined values based on their parameter types.

In 2010, Trinity [25] extended the idea by adding some randomness for the generation of test cases. For example, it selects a random integer value for a length parameter, whereas Koopman used only eight predefined integer values. Moreover, Trinity does not exhaustively test all possible combinations of parameter values, because there are simply too many possibilities due to the randomness. Instead, it randomly selects a value for each parameter during a fuzzing campaign. For example, whenever it encounters a `read` system call, where the first parameter is a file descriptor, it randomly selects one from a pool of valid file descriptors initiated when fuzzing starts. It also generates the remaining parameter values in a similar fashion. KernelFuzzer [33], Chen *et al.* [9], Syzkaller [53], iknowthis [44], TriforceLinuxSyscallFuzzer [40], and perf_fuzzer [55] are in this category. We call such fuzzers *type-aware kernel fuzzers*. Type-awareness appears in most recent kernel fuzzers including [17, 21]. For example, Eris [17] improves upon Trinity by adapting the concept of combinatorial testing when selecting parameter values.

Hooking-based Kernel Fuzzers. Some fuzzers attempt to fuzz the API calls by intercepting API function calls while running a program. We call such fuzzers *hooking-based kernel fuzzers*. IOCTL Fuzzer [42] specifically aims to fuzz IOCTL requests of Windows by hooking a single API call, `NtDeviceIoControlFile`, randomly mutating its parameter values at runtime. Similarly, IOKit fuzzer [4] hooks a single IOKitLib function, `IOConnectCallMethod`, which can be considered as a counterpart of `NtDeviceIoControlFile` on Mac. PassiveFuzzFrameworkOSX [36] hooks API functions in XNU kernel with a custom kernel driver. Fuzzers in this category do *not* generate concrete test cases. That is, even if they found a kernel crash, they do *not* produce the corresponding program that triggers the bug when it runs. This is the key difference between traditional mutation-based fuzzing [8, 56] and hooking-based fuzzing. Furthermore, hooking-based fuzzers always follow error-handling routines of a program, which can limit their testing scope (see §3.1).

Feedback-driven Kernel Fuzzers. Some kernel fuzzers such as Syzkaller [53], kernel-fuzzing [43] and TriforceLinuxSyscallFuzzer [40] leverage code coverage during the generation of system calls. Specifically, they choose one from among randomly generated system calls that maximizes the code coverage. The same principle is indeed used in state-of-the-art userland fuzzers such as AFL [60] and honggfuzz [50]. Similarly, perf_fuzzer [55] repeatedly calls a system call with random parameters until the call succeeds. That is, it uses the return code from a function as feedback. Lei [37] presents an approach similar to that of perf_fuzzer for fuzzing I/O kit on iOS. We refer to the fuzzers in this category as *feedback-driven kernel fuzzers*.

Our Contribution. None of the kernel fuzzers discussed thus far takes calling contexts into account. One notable exception is [19]. Gauthier *et al.* propose the use of a manually constructed API model. However, they did not implement nor evaluate their approach, and the approach relies on manual construction of an API model. To the best of our knowledge, IMF is the first practical attempt to realize model-based kernel fuzzing.

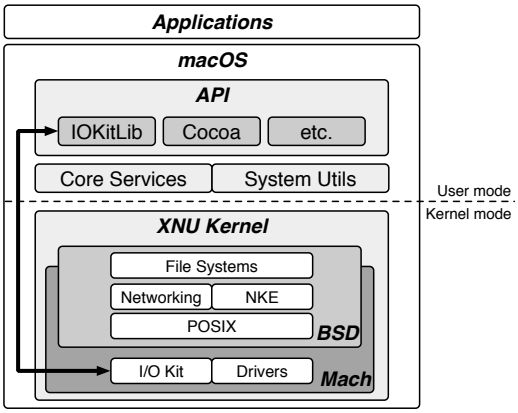


Figure 2: XNU Kernel architecture.

2.3 Mac OS and its Kernel

Since the codename Sierra, the name of Apple's OS for desktops and servers has become "macOS", which was previously known as "Mac OS", "Mac OS X", or "OS X". Unless otherwise specified, the term macOS in this paper means the Sierra version of Apple's OS, which is the latest version at the time of this writing.

XNU is the kernel of macOS, which consists of three major components: Mach, BSD, and I/O Kit [22, 35]. Both Mach and BSD form the basis of the XNU kernel. The Mach component is responsible for various low-level tasks such as processor management, interrupt management, and inter-process communication. The BSD component of XNU handles higher-level operations compared to Mach such as file system and network management. It also provides APIs for POSIX and system calls. I/O Kit is a framework for device driver development and management. It provides an abstract view of the system hardware with an object-oriented programming model. IOKitLib is a set of API functions provided with I/O Kit. A user program can access a device with functions in IOKitLib. Figure 2 summarizes the architecture of XNU. IOKitLib itself runs in user mode, but it invokes functions in the kernel and drivers.

I/O Kit is currently a popular hacking target. Hackers attempt to jailbreak iOS by exploiting I/O Kit vulnerabilities [38]. They write fuzzers that specifically target IOKitLib functions [4, 21, 37]. By fuzzing the IOKitLib functions, we can directly find bugs in device drivers, which typically run in privileged mode, as well as in the kernel (I/O Kit) itself. We emphasize again that none of the previous I/O Kit fuzzers consider API specifications. Our main fuzzing target is also IOKitLib API functions. However, the proposed idea is general enough to be applied to any kind of API function such as Linux system calls and the Win32 API of Windows.

3 OVERVIEW

The primary goal of IMF is to automatically determine an API model, and to use the model effectively to fuzz the kernel. In this section, we present an example that motivates our research. Next,

```
1   uint64_t inScalar[0x10];
2   char inStruct[0x1000];
3   uint64_t outScalar[0x10];
4   uint32_t outScalarCnt = 0x0;
5   char outStruct[0x1000];
6   size_t outStructCnt = 0x0;
7   io_iterator_t iterator;
8   io_connect_t conn;
9   io_service_t service;
10  CFMutableDictionaryRef r = IOServiceMatching("IntelAccelerator");
11  IOServiceGetMatchingServices(kIOMasterPortDefault, r, &iterator);
12  service= IOIteratorNext(iterator);
13  IOServiceOpen(service, mach_task_self(), 0x1, &conn);
14  IOConnectCallMethod(conn, 0x205, inScalar, 0x0, inStruct, 0x30,
15      outScalar, &outScalarCnt, outStruct, &outStructCnt);
16  IOConnectCallMethod(conn, 0x206, inScalar, 0x0, inStruct, 0x1,
17      outScalar, &outScalarCnt, outStruct, &outStructCnt);
```

Figure 3: An example program that exploits a kernel vulnerability (CVE-2015-7077) using IOKitLib functions.

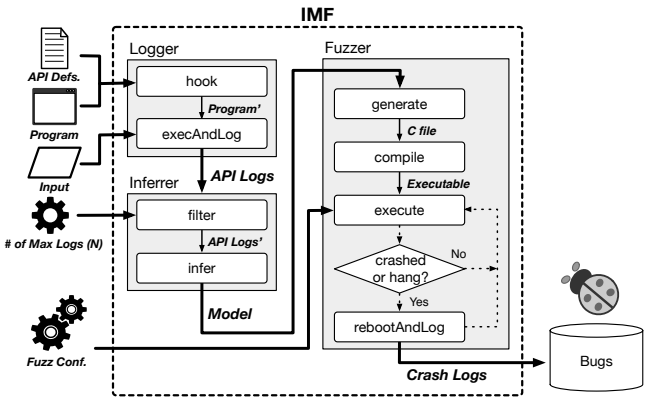


Figure 4: IMF Architecture.

we show the overall architecture of IMF. Finally, we describe several challenges in the design of IMF.

3.1 Motivation

To see why understanding an API model is important in API fuzzing, we initially present how IOKitLib functions are used in the wild. Figure 3 shows a code snippet taken from a recent macOS kernel exploit [39]. The `IOServiceMatching` function in Line 10 creates a dictionary that can match Intel's graphics drivers. The function `IOServiceGetMatchingServices` then returns a pointer (iterator) to a list of matching drivers. In Line 12, we obtain a handle for the first matching driver from the iterator. Next, we open a connection to the device driver with `IOServiceOpen` in Line 13. The second argument of the function is a kernel port, which indicates who is requesting the connection. The `mach_task_self` function returns the caller's kernel port. The third argument of the function specifies the connection type. Here, type 1 indicates that we open a connection to the `IGAccelGLContext` interface of the driver. The function returns the handle of the interface to the variable `conn`.

Finally, we make two calls to `IOConnectCallMethod`, which takes in ten arguments as input. The first argument is the connection handle (`conn`), and the second is a selector specifying which method to call. The remaining arguments specify the parameters to be used in the method. The third argument is a pointer to an array of scalar (64-bit number) input values, and the fourth argument is the size of the array. The fifth argument is a pointer to a C struct input, and the sixth argument is the size of the struct. Similarly, the last four arguments denote output parameters to the method. In Lines 14 and 15, we invoke the `gst_operation` method (`0x205`) and the `gst_configure` method (`0x206`) of `IGAccelGLContext`, respectively, with the input/output parameters prepared on the stack. The vulnerability is at Line 16, where the second call to `IOConnectCallMethod` uses a small struct size (`0x1`). Given that `gst_configure` does not check the size of the input struct, we can trigger an out-of-bounds memory access.

This example highlights the difficulty of triggering vulnerabilities by invoking API calls. The vulnerability is triggered only when we invoke the API functions in a specific order with a specific set of values. In this paper, an *API model* specifies (1) in which order API functions should be called, and (2) which value should be used for each parameter in each function.

In this paper, an API model includes two types of dependences between function calls. The first is an *ordering dependence*, which indicates that a call to function A should always precede a call to function B. The second is a *value dependence*, which shows the relationship between an output of a call to A and an input of a call to B. Indeed, a similar intuition is found in the literature on malware behavior analysis [11], where their goal is to extract behavioral features from API call sequences.

Ordering Dependence. The ordering of function calls matters. From the previous example, a call to `IOConnectCallMethod` should appear after a call to `IOServiceOpen`, as there must be an open connection to a device driver in order to invoke a method in the driver. We call such dependence as *ordering dependence*. We denote as $A \prec B$, when there is ordering dependence between A and B: A should precede B. The two function calls to `IOConnectCallMethod` in Figure 3 also have ordering dependence: if the order between the two calls changes, the vulnerability is not triggered because the first call initializes an internal data structure that is used by the second call.

Value Dependence. Most API functions take in data returned from another API function. When a returned value of function A is used as input to a call to function B, we say that two functions have value dependence. For example, in Figure 3, `IOConnectCallMethod` takes in as the first argument a connection handle that is returned from `IOServiceOpen`. Note that in Figure 3, we had to call a series of API functions (from Lines 10 to 14) in order to trigger the vulnerability. However, simply invoking the same sequence of API functions with random parameter values does not help in this case, because most functions are dependent on other functions. Without considering the dependence relationship, it is unlikely to find such an instance of vulnerability.

3.2 Architecture

We introduce IMF, a system that automatically finds kernel API vulnerabilities with inferred API models. At a high level, IMF automatically determines both ordering and value dependences between

API logs, which we call an *API model*, from program executions and uses it to fuzz the target API functions. IMF consists of the three major components including a logger, an inferrer, and a fuzzer. Figure 4 describes the overall architecture of IMF.

Logger. The logger takes in as input a list of definitions of target API functions, a program, an input to the program, and the number of logs to take (L). It returns a list of L API logs obtained by executing the program L times with the given input. Figure 4 does not present the parameter L for simplicity. If not given, we use $L = 1,000$ by default in our implementation. The definitions of API functions are prototypes with annotations, which describe how each function uses its parameter. That is, an annotation of a parameter specifies whether the parameter is used as input, output, or both. We assume that such definitions are given by an analyst. The input to the program is a combination of a file, a command line argument, mouse events, etc. The `hook` function installs API hooks in the program, and outputs a modified program. The `execAndLog` function executes the modified program L times and stores the history of every hooked function call along with the parameter and the return values for each execution (see §4.1).

Inferrer. The inferrer figures out the API model, i.e., both ordering and value dependences between API calls, from the L API logs returned by the logger. Specifically, it takes in as input the maximum number of logs to use N and the L API logs, where $N \leq L$, and outputs an inferred model. First, the `filter` function of the inferrer selects N logs from the L API logs, which have the longest common prefix. The returned logs are then used to infer the model in the `infer` function to determine the ordering and value dependences (see §4.2).

Fuzzer. The fuzzer module takes in an inferred API model from the inferrer and a fuzzing configuration as input, and outputs bugs found along with the test cases. The `generate` function converts the inferred model into a C file. IMF compiles this C file into an executable. The `execute` function takes in the compiled executable program and the fuzzing configuration as input, and iteratively runs the program to find bugs. When a kernel panic occurs, the `rebootAndLog` function of IMF checks if the corresponding crash log exists. If so, it stores the log in our bug database. The fuzzer module then resumes the fuzzing process. This process repeats until it reaches a timeout (see §4.3).

3.3 Running Example

To describe the overall procedure of IMF, we use an application called "2048 Game" as our running example, which was the second most popular game in App Store as of March of 2017. Suppose we run IMF with five inputs: (1) a header file that defines the prototypes of IOKitLib API functions with annotated parameters; (2) the 2048 Game application; (3) a series of mouse clicks as input to the program, generated by a small script that we wrote; (4) the maximum number of logs to take (N), which is 2 in this example; and (5) the fuzzing configuration, which essentially describes how API calls are fuzzed. IMF works via the following steps.

First, IMF installs API hooks in the program based on the given header file prior to executing the program. An analyst can declare any functions to hook in the header file, but for this example, we assume that we are targeting all of the IOKitLib functions. IMF runs the program with the given input 1,000 times, i.e., $L = 1,000$. For each hooked function call, the logger of IMF refers to the corresponding function definition to check the type and the attribute of the parameters. For example, it checks whether a parameter is used as input, output, or both. It then returns an API log for each execution of the program, which is essentially a list of input and output values for each function call encountered during the execution. In the end, the logger results in 1,000 API logs.

Due to the non-deterministic nature of GUI events, the 1,000 API logs are mostly different from each other. In this example, IMF selects $N(= 2)$ logs that contain the longest common prefix. We detail how we filter out the logs in §4.2. Figure 5a shows two example snippets of API logs generated from the application. Note that both logs have exactly the same sequence, but their parameter values differ.

Now that we obtained the subset of the API logs, the inferrer module of IMF infers both ordering and value dependences between the API calls in them. First, we assume that the order of the API calls in the log should be preserved. This gives us over-approximated ordering dependences between the API calls, because we may include unnecessary ordering dependences in our model. However, we relax this assumption when we fuzz API calls. The intuition is that bugs frequently occur when a random-yet-legitimate-looking API sequence is used. From the example, we obtain the following ordering dependences:

(1) `IOServiceMatching` $\prec$ `IOServiceGetMatchingService`
(2) `IOServiceGetMatchingService` $\prec$ `IOServiceOpen`
(3) `IOServiceOpen` $\prec$ `IOConnectCallMethod`

Next, the inferrer finds constant function parameters that are always identical across all the subset of the logs. Because it knows from the API definition that which parameter has a handle type, e.g., `io_service_t` and `io_connect_t`, it does not count such a parameter as a constant (see §4.2.2). In Figure 5a, the values in green boldface indicate constant parameters. With this information, the inferrer searches for a pair of the same input and output values in the log. However, it excludes input values that are constant. In this example, the return value of the first function call (`0xd32e0a90`) is the same as the second parameter of the second function call. Thus, we say those two parameters have a value dependence. However, even though the third argument of the fourth function call (`0x0`) has the same value as the return value of the third call, we do not say there is value dependence between them, because the input value is a constant. We denote the value dependences by dashed arrows in Figure 5a. IMF employs several heuristics to improve the accuracy of the dependence inference, which we describe in §4.2.

Combining both ordering and value dependences along with the API definitions given by an analyst, IMF automatically generates a model. Figure 5b presents the derived API model from our example in C language. In our actual implementation, we use an Abstract Syntax Tree (AST) to represent the API model. Each function call in the model follows the order of the original logs in Figure 5a. It also shows clear value dependences between functions with the use of variables. Constant values are directly used in the model, as in Line 7 and 10. The relationship between a pointer to a structure and the length of the structure is also represented in Line 8 and 9.

```
IOServiceMatching("IOSurfaceRoot")
└─0xd32e0a90 ----------------------┐
IOServiceGetMatchingService(0x0, 0xd32e0a90)
└─0x10127 ------┐
IOServiceOpen(0x10127, 0x103, 0x0, out)
└─0x0, out:[0x10203]---┐
IOConnectCallMethod(0x10203, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
                    out, inout:[0x14])
└─0x0, out:[0x3f, ...], inout:[0x14]
```

```
IOServiceMatching("IOSurfaceRoot")
└─0x31177280 ----------------------┐
IOServiceGetMatchingService(0x0, 0x31177280)
└─0x10327 ------┐
IOServiceOpen(0x10327, 0x103, 0x0, out)
└─0x0, out:[0x10403]---┐
IOConnectCallMethod(0x10403, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
                    out, inout:[0x14])
└─0x0, out:[0x3f, ...], inout:[0x14]
```

(a) Two API logs. A solid arrow shows a return value from a function, a dashed arrow indicates a value dependence.

```
1  io_connect_t conn;
2  io_service_t service;
3  size_t outStructCnt;
4  void* outStruct;
5  CFMutableDictionaryRef r = IOServiceMatching("IOSurfaceRoot");
6  service = IOServiceGetMatchingService(0x0,r);
7  IOServiceOpen(service,0x103,0x0, &conn);
8  outStructCnt = 0x14;
9  outStruct = calloc(outStructCnt,0x1);
10 IOConnectCallMethod(conn, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
11                    outStruct, &outStructCnt);
```

(b) An inferred model from the logs in C.

Figure 5: An example from an app. called "2048 Game".

The `outStruct` variable is a pointer to a structure the size of which is `outStructCnt`. IMF retrieves such relationships from the API definitions given by an analyst and represents them in the model.

The fuzzer module of IMF finally generates a C program from the API model. In the final C program, we replace every parameter in the model with a function call that returns a random value of a specific type. For example, we change Line 5 of Figure 5b to

```
CFMutableDictionaryRef r =
    IOServiceMatching(mutString("IOSurfaceRoot"));
```

where `mutString` is a function that returns a randomly generated string pointer from a given string pointer and a fuzzing configuration. The fuzzing configuration is a set of user-configurable parameters that includes a PRNG's seed. Thus, the function always returns the same value if the same seed is used. We define mutation functions for several different types as well as the fuzzing configuration in §4.3.

The fuzzer module of IMF is responsible for executing the generated program. For every iteration, IMF simply changes the value of the PRNG's seed, and run the generated program, which will invoke a series of API functions with randomly mutated parameter values. When the OS hangs or crashes with a kernel panic, IMF checks the corresponding crash dump. If the dump exists, IMF stores it in the bug database along with the PRNG's seed value. This process repeats until a timeout is reached.

4 DESIGN

In this section, we describe the design of IMF in detail. For each module we described in §3.2, we present our design decisions and several challenges we faced while implementing our system.

4.1 Logger

We address two challenges in designing the logger. First, we need to consider how much data we should record from the logger. Second, we should generate inputs to GUI applications in order to log the API calls.

4.1.1 Amount of Data to Log. The logger records the input and output values for each hooked API function call. However, it is not clear how many levels of pointer indirection we need to deal with. One needs to follow all the pointers to store all data completely, but this imposes a significant space overhead. In our current implementation, we only store the data up to one level of pointer indirection. We believe that our inferrer can infer more accurate dependences as we log more data, but we leave it as future work to handle additional levels of indirection.

When a hooked API call is invoked, the logger examines the type of parameter, and decides which values to log. When the type of a parameter is a pointer, we store both the value of the pointer and the corresponding dereferenced value. When the pointer is a character pointer, we log the corresponding C string, i.e., an array of characters that ends with a NULL character.

4.1.2 Input Generation. Most applications used in this paper are GUI applications. Because GUI applications typically listen for keyboard and mouse events as input, we need to have a way to automatically generate such inputs in order to run them repeatedly. We use PyUserInput [3] to construct mouse events for each program that we are running. Suppose we are targeting a game application. We first manually outline a series of mouse clicks required to play the game. We then compute specific X and Y coordinates for each of the mouse clicks, and write a script using the PyUserInput library. To handle more complicated interactions, one may use an advanced GUI automation tool such as Sikuli [59]. However, we note that our focus is on fuzzing, and not on taking accurate API logs. Thus, incorporating such a tool in IMF is beyond the scope of this paper.

4.2 Inferrer

Recall from §3.2, the inferrer module mainly performs two steps. We explain how we filter out API logs. Then, we describe our algorithm of API model inference.

4.2.1 Log Filtering. API logs taken from the same program with the same input can contain largely different sequence of API function calls with distinct input/output values. Pointers to heap or stack objects can change over executions due to ASLR. Multi-threaded applications as well as event-driven applications can result in different call sequences due to its non-deterministic nature.

The filter function of the inferrer selects N logs that have the longest common prefix from the given the entire set of logs. It then collects the prefix part of each of the selected logs in order to construct a set of call sequences S that have exactly the same order and the same number of of API calls. We note that even though

the call sequences in S have the same order, they are likely to have different parameter values due to the non-determinism, which gives us new opportunities to figure out the API model.

4.2.2 API Model Inference. We determine two types of dependences in the inferrer: ordering and value dependence. First, the call sequences in S already show the ordering dependences between API function calls. Since the same order of API calls appeared $N = |S|$ times in the legitimate execution of the same program with the same input, we can over-approximate the ordering dependences by considering that the ordering of API calls should follow the exact same sequence as in one of the sequences in S.

To compute the value dependences, we first identify constant input parameters from the sequences. Given N different call sequences in S of the same size, there can be input parameters that have the same value across different sequences. When an input parameter of a function, i.e., a parameter that is annotated as an input in the definition of the function, always has the same value in N different call sequences, we conclude that such parameter values are used as a constant, because input variables are likely to change across executions if they are returned from a function. We call such an input parameter as a *constant input parameter*.

Let $s^k_{i,j}$ be the ith parameter of the jth API call in the kth call sequence in S. If there exists i and j such that $s^1_{i,j} = s^2_{i,j} = \ldots = s^N_{i,j}$, then we say the parameter $s_{i.j}$ is a constant input parameter. For example, the second and the third parameter of open system call are constant input parameters, because their value never changes even though we execute the program multiple times with the same input. However, the first parameter of open can vary over multiple executions, if the string is dynamically allocated. We employ a type-based heuristic to improve the accuracy of our analysis. In particular, we exclude handle types when judging a constant parameter, because handle types, e.g., a file descriptor, may have the same value across multiple executions even though they are not actually a constant.

Next, we consider approximated data flows from an output parameter to an input parameter. For a given input parameter value in a sequence $s \in S$, we check if there are any prior function calls in s that have the same output value. If so, we say that the input parameter is *value-dependent* on the output parameter. When an input parameter is value-dependent on multiple output parameters, we take a parameter that is used in the most recently invoked function. The key rationale here is that recently used variables are more likely to be live [1].

Improving the Precision. Of course, the above approach can result in false value dependences, because it relies only on the parameter values. To reduce such false dependences, the inferrer employs the following two techniques. First, it excludes data flows to a constant input parameter. Intuitively, a constant input parameter does not get its value from an output of a function. Second, we derive sets of value dependences on each of the N call sequences, and compute the intersection of them. Each call sequence can exhibit different dependence relationships due to dynamically changing parameter values. By taking the intersection of the sets, we can potentially remove false dependences. The primary intuition of both techniques is that multiple API logs from the same program execution can help in inferring data flows between parameters. We evaluate the effectiveness of these techniques in §5.2.

The Output. The final output of the inferrer is an Abstract Syntax Tree (AST) of a C program. The ordering dependences naturally lead to a series of function call statements in the specific order. The parameters in each of the function calls are filled with constant values and variables. For constant input parameters, we use the constant values appeared in the log. For non-constant input parameters that have a value dependence, we declare a variable of the parameter type, and connect the corresponding output parameter with the input parameter using the variable. Suppose the first parameter of a function "`void B(int b)`" is value-dependent on the first parameter of a function "`void A(int* a)`", and the type of the parameter is `int`. Then, we create an AST node that comprises the following statements:

```
int a_0;
int b_1;
A(&a_0);
b_1 = a_0;
B(b_1);
```

Each variable name is suffixed with a number that indicates the order of appearance. Finally, for those non-constant input variables that do not have any dependence relationship, we simply use the same concrete value appeared in the log. Since there are N possible values to use, we take one of the values at random.

4.3 Fuzzer

The fuzzer module in IMF takes in an API model (in an AST) and a fuzzing configuration as input, and generates a C program that can fuzz the kernel API functions in the model. The key question that we need to answer is: how can we generate a C program from the given AST in such a way that it can fuzz the kernel API functions every time it runs.

4.3.1 Fuzzing Configuration. A fuzzing configuration comprises five user-configurable parameters: (1) T is a timeout, (2) I is the number of iterations, (3) P is a mutation probability, (4) F is the number of fixed bits, and (5) R is a PRNG's seed. IMF iteratively runs the final C program until the given timeout reached. The program takes in the fuzzing configuration parameters as command line arguments. The number of iterations I specifies how many times we should repeatedly call API functions. Both mutation probability P and the number of fixed bits F are used in mutating parameter values (§4.3.2). A PRNG's seed R is an optional argument that is only used when we reproduce bugs found. If it is given by an analyst, the program initializes its PRNG with R. Otherwise, it initializes the PRNG with the system time. This is to help in deterministically reproducing test inputs we used in each fuzzing iteration. We discuss how we store the seed values in §4.3.3.

4.3.2 Mutation Strategy. IMF employs two simple mutation strategies. First, it replicates a given API call sequence of the model. Second, it randomly mutates parameter values based on the given model. We use the same intuition as in traditional mutational fuzzing: we mutate only a subset of the parameters at random, because mutating every parameter can easily break the dependence

relationships between the API calls. Namely, fuzzing should generate a random, but legitimate-looking input, which is a function call sequence in our case.

Sequence Replication. The obtained call sequences from the logger may contain only a few number of API calls, e..g, some logs only contain 5 API calls in our dataset, depending on the program and the input used. To extend the number of API calls to fuzz while preserving the ordering dependence, we replicate the entire function calls within a for loop, where the number of iterations I is determined by an analyst. In our experiment, employing this strategy helps in finding higher number of kernel bugs (§5.3).

Parameter Mutation. IMF mutates each parameter value based on the mutation probability P and the number of fixed bits F. The key design principle here is to mutate parameter values in such a way that the mutated values are similar to the original. The mutation probability specifies for each parameter the probability of mutating its value. For example, when $P = 0.01$, we mutate each parameter 1% of the time. The number of fixed bits F specifies the number of higher bit positions IMF should *not* mutate. For example, when $F = 20$, IMF will mutate only the lower 12 bits of an `int32_t` parameter by XORing the lower 12 bits of the parameter with a 12-bit random number. When $F = 0$, IMF will simply replace the whole parameter value with a random number.

IMF performs type-based parameter mutation. It replaces each parameter in the AST with a call to a mutation function. We define mutation functions for each primitive type including `char`, `short`, `int`, etc.. Each mutation function takes in the original value in the model, and mutates the value based on P and F. We note that F may exceed the size of a certain type. For example, when $F = 16$, a mutation function for `int8_t` has a smaller number of bit positions to mutate compared to F. In such a case, we only mutate the LSB of the original parameter value. In case of array types, we call a mutation function for every element in the array.

4.3.3 Data Collection. Unlike userland fuzzers, IMF may trigger a kernel panic or a system hang when it finds a bug. Thus, it must save the current value of the PRNG's seed into permanent storage prior to the execution of the program. Otherwise, we may lose the last parameter values used to crash the kernel. Specifically, IMF stores the current seed value in a status file located on a configurable path. To detect a system hang, IMF employs a watchdog. It periodically checks the last modified time of the status file, and if the file is not updated for more than five minutes, it forcefully reboots the OS. When the OS reboots, IMF checks the last panic log and stores the corresponding PRNG seed into our bug database along with the corresponding crash dump.

4.4 Implementation

We have implemented IMF with 1.2K lines of Python code. We used PyUserInput [3] library for a keyboard and mouse automation. To speed up the model inference, we used multiprocessing libraries in Python. IMF currently relies on manually constructed API definitions, which is written in Python. To compile the generated C file, we used an LLVM compiler (8.1.0) provided by Xcode 8.3.

To write a log file, the logger needs to consider the App Sandbox, which is an access control mechanism used in macOS. Specifically, it restricts the access of applications to files, network connections, or a hardware component such as a camera or a microphone. In order to use such a resource, developers must specifically request access to the OS. For our purpose, we need to check in which directory we have a permission to write files, because the logger must write a log file to disk. To this end, IMF determines a writable directory path by parsing an information property list file (`Info.plist`), which stores app-specific metadata. Any sandboxed apps have full read/write access to its container directory, and hence the logger does also.

Finally, we make our source code public on Github to boost future research towards effective kernel fuzzing: https://github.com/Softsec-KAIST/IMF.

5 EVALUATION

We now evaluate IMF on macOS to answer the following questions:

(1) Can IMF infer accurate API models? (§5.2)
(2) Does mutation configuration affect the effectiveness of fuzzing? (§5.3)
(3) How does IMF perform compared to the existing kernel fuzzers? (§5.4)
(4) Can IMF find realistic vulnerabilities? (§5.5)

5.1 Experimental Setup

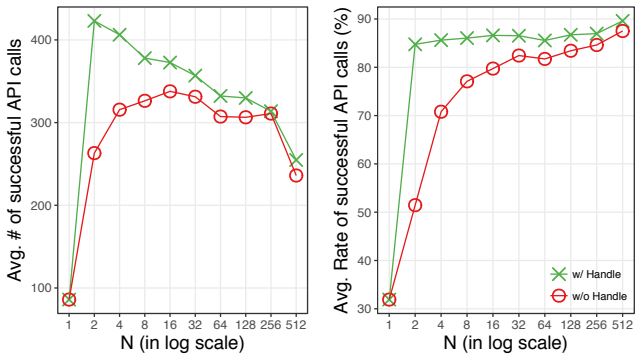
We ran a series of experiments on 10 Mac minis (late 2014). Each machine was running macOS version 10.12.3 (build 16D32), which was introduced in Jan. 23, 2017, on Intel Core i5 2.6 Ghz CPU with 8 GB of memory. The number of bugs reported from this paper is based on manual investigation of crash dumps.

API Log Collection. To gather API logs, we first downloaded 105 apps from the US App Store: 5 apps per each of the 21 total categories. We selected 5 most popular and free apps in each category as of Mar. 1st, 2017. We then manually set up a script for each of the applications to generate GUI inputs as we described in §4.1.2. We then ran the logger 1,000 times for each application with the definitions of 93 IOKitLib functions. As a result, we successfully gathered logs from the 95 out of 105 applications. We could not obtain logs from the 10 applications for various reasons: some programs require specific H/W devices such as Camera, which is not present our Mac minis; one of the programs uses anti-hooking techniques; some other programs do not use IOKitLib functions at all. Thus, in total, we obtained 95,000 logs from the 95 applications. The average number of API calls in each of the logs was 687. Our evaluation makes use of this dataset throughout this section.

5.2 API Model Accuracy

Does it really help to have multiple logs in terms of the accuracy of inferred API models? If so, how many API logs (N) should we take? We measured the accuracy of inferred models to answer these questions.

We first ran the inferrer on the 95,000 collected logs in order to get 95 API models and the corresponding C programs. We then ran each of the 95 programs without mutating the parameters nor replicating the sequences, i.e., we set $P = 0$ and $I = 1$, in order to measure the precision of the API models. We then counted the followings: (1) the number of API calls that return zero, which



(a) # of successful API calls. **(b) The success rate of API calls.**

Figure 6: The comparison of API model accuracy w/ and w/o considering handle types.

means "success"; and (2) the rate between the number of successful API calls and the total number of API calls. Figure 6 shows two diagrams depicting the accuracy of the models over N.

5.2.1 Does having multiple logs help? The crux of our system is that we can increase the precision of the inferrer by having multiple logs. To justify the intuition, we compare the precision of derived API models with and without having multiple logs. We computed the number of successful API calls over N in Figure 6a. When $N = 1$, the number of successful API calls was only 86 on average. However, when $N = 2$, the number became 263, that is, the precision was improved by 300%. As we increase N more, we get less successful API calls, because the size of the common prefix between N logs decreases. To better understand the API model accuracy, we measure the success rate between the number of successful API calls and the total number of API calls. Figure 6b shows the success rate over N. When we take 512 out of 1,000 logs, we could achieve the 90% success rate. This graph clearly indicates that we always get better accuracy as we increase N, the number of logs to consider.

Recall from §4.2.2, IMF relies on the handle type information when it identifies constant input parameters. Since this is an important step in our inference technique, we also check the precision of our analysis with and without having a knowledge about handle types. Both diagrams in Figure 6 show that considering handle types gives better accuracy. Furthermore, the same intuition still applies in this case: as we have more number of logs, we can increase the precision of our API model inference in terms of the success rate.

Notice, our accuracy measure can have an error: there can be API calls that return success even though the model is incorrect, or vice versa. However, since we are averaging the result of 1,000 iterations, the false success rate should be marginal. We also manually checked the accuracy of API models that result in kernel panics, and found that our model was indeed accurate.

5.2.2 Which value of N should we use? From the above result, it is clear that as we compare more number of logs altogether, we can obtain more precise API models. However, as it is shown in Figure 6a, the total number of API calls gets decreased as we increase N. This is mainly due to the difficulty of having multiple logs while preserving the size of the longest common prefix: as we increase the number of logs to consider, we get the shorter common prefix between them. Since our focus is on fuzzing, but not on inferring the API model itself, it is important for us to have an enough number of API calls. Taking this trade-off into account, we select $N = 2$ in our experiments in order to maximize the number of API calls to use while maintaining the reasonable success rate (84.8%).

5.3 Mutation Configuration

IMF takes in three user-configurable parameters as input to decide the mutation strategy (§4.3.1). We evaluated the effectiveness of each parameter by running IMF with several combinations of the parameters. Specifically, we considered the following parameter values and their combinations:

(1) Number of iterations (I): 1, 10, 100, 1000.
(2) Mutation prob. (P): 1/10, 1/100, 1/1000, 1/10000, 1/100000, 1/1000000.
(3) Number of fixed bits (F): 8, 12, 16, 20.

We tested all 96 ($= 4 \times 6 \times 4$) combinations on a subset of the models in our dataset. Due to resource limit, we chose only the five API models taken from the five apps in the game category when $N = 2$. We selected game applications because they are largely dependent on I/O Kit functionalities such as audio and graphics. We fuzzed macOS for 480 hours in total with the five API models and with different combinations of the parameters: we ran 1 hour of fuzzing for each model and each combination of the parameters.

Figure 7 summarizes the result. Each row in the figure shows the number of unique kernel panics found, the number of userland crashes, and the number of kernel hangs, respectively. First, we note that the number of bugs found largely depends on the choice of I, the number of iterations. We found clearly a larger number of bugs when I is 100 or above. Second, the mutation probability also significantly affects the numbers. As we increase the probability, we could find more kernel panics or crashes. However, if the probability was too large, i.e., when $P = 1/10$, then the number drastically decreased. This makes sense because it is highly likely that we will break the dependences as we mutate more number parameters. Indeed, the same intuition has been observed in userland mutational fuzzing [8]. Finally, the number of fixed bits (F) did not affect much the result from our dataset.

We conclude that a mutation configuration can affect the effectiveness of kernel fuzzing. In our dataset, the combination of parameters $I = 1,000$, $P = 1/1000$, and $F = 20$ achieved the most effective fuzzing results in terms of the number of kernel panics found. Thus, we are going to use this parameter combination in the rest of the experiments.

5.4 Comparison against IOKit Fuzzer

How does IMF compare with existing kernel fuzzers? To answer this question, we compared IMF against IOKit Fuzzer [4] released by Google Project Zero in 2014. We selected this fuzzer, because it is open-sourced and is specifically targeting IOKitLib functions. Furthermore, it has proven to be effective in finding kernel vulnerabilities: it has already found numerous CVEs. To run IMF, we

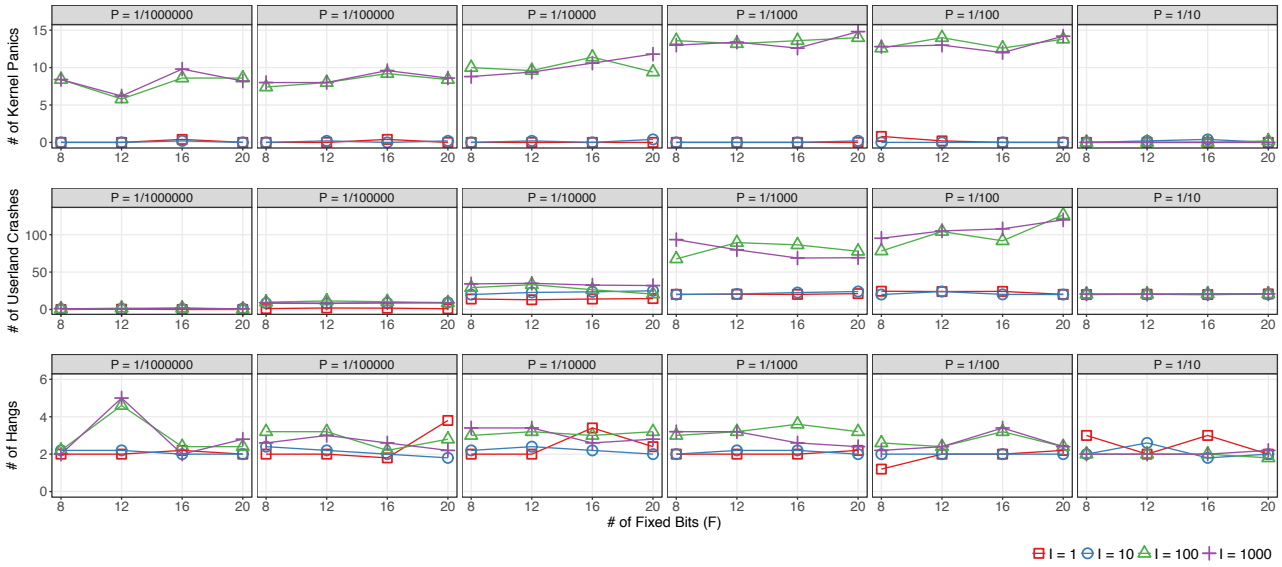


Figure 7: # of bugs in each mutation configuration.

selected the five API models we used in §5.3, and chose the parameters as $I = 1,000$, $P = 1/1000$, and $F = 20$. We used the same five game applications to run IOKit Fuzzer. Since it is a hooking-based kernel fuzzer, we also used the scripts described in §5.1 to generate GUI inputs while running it.

We ran both fuzzers for 24 hours on macOS, and counted the number of unique kernel panics found. As a result, IOKit Fuzzer found 3 unique kernel panics, whereas IMF found 10: IMF discovered 3× more kernel bugs than IOKit Fuzzer, but there was no overlap between them. And crash points of bugs were different. In case of IOKit Fuzzer, crash points were in only IOKit Fuzzer thread. But in case of IMF, crash points were in IMF, `reboot`, `mdworker`, `ReportCrash` and `mds_stores` thread. We note that the main difference between the two fuzzers is that IOKit Fuzzer always follows the execution of an application due to the nature of a hooking-based kernel fuzzer. Whenever an API call returns an error, a target application typically handles the error, e.g., by exiting the function or the program. On the other hand, IMF will keep execute a series of API functions regardless of their return code. We believe this difference gives IMF an opportunity to explore deep in the kernel code to find latent bugs.

5.5 Large-Scale Bug Finding

Can IMF find realistic vulnerabilities? We answer this question by fuzzing macOS at large-scale. We ran fuzzing on macOS with 95 API models in our dataset for 1,140 hours in total, 12 hours for each API model. The macOS version was Sierra 10.12.3, which was the latest at the time of writing. As a result, IMF found 2,017 kernel panics and 14,703 userland crashes in total. We manually investigated the corresponding crash dumps of the kernel panics, and categorized them into 26 unique kernel panic vulnerabilities. None of the vulnerabilities found were previously known.

Figure 8 shows the number of unique kernel panics found over time in our experiment. This graph combines all of the fuzzing results from each of the 95 API models into a single timeline: this is the reason why the X-axis of the graph ranges from 0 to 12 hours. Figure 9 presents the number of unique kernel panics found from API models derived from each category on App Store. At least one model in each of the 21 categories resulted in a kernel panic. One thing to note is that the API models derived from the game category produced the most unique number of kernel panics. It may be the case that our choice of mutation parameters is optimized for the game applications (in §5.3). However, we leave it as future work to study the correlation between the choice of parameters and the effectiveness of IMF.

We also measured how many IOKitLib functions out of the 93 total are included in the inferred models for each category. Figure 10 shows the coverage of API functions used in models of each category with error bars. The green line indicates the averaged coverage of all the API models, which was 19.1%. From the result, we could *not* identify a clear correlation between the API coverage of a model and the number of unique kernel panics. For example, the 'News' category covered the most number of API functions, but the number of unique kernel panics found was below the average (from Figure 9).

We believe that all the vulnerabilities that we found have a security impact because they can cause at least a Denial-of-Service (DoS) attack on the entire OS. In the worst case, they may allow attackers to gain unprivileged root access.

5.6 Case Studies for the Bugs Found

We now discuss all the bugs that IMF found during the experiments in §5.3, §5.4, and §5.5. Combining all the three experiments, we found 32 unique kernel panics including General Protection Fault (GPF), NULL pointer dereference, and kernel object corruption, as

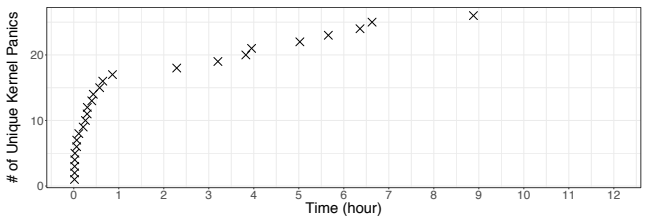


Figure 8: # of unique kernel panics found over time.

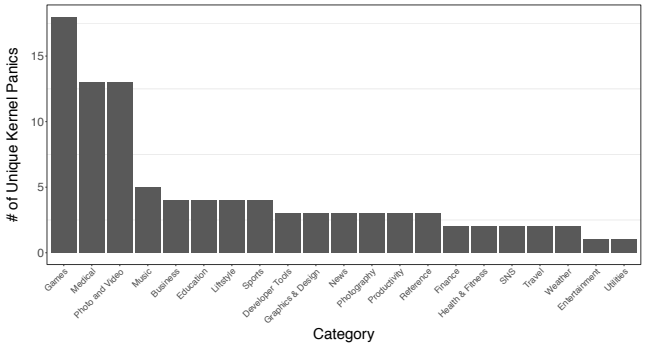


Figure 9: # of unique kernel panics found from API models derived from each category on the App Store.

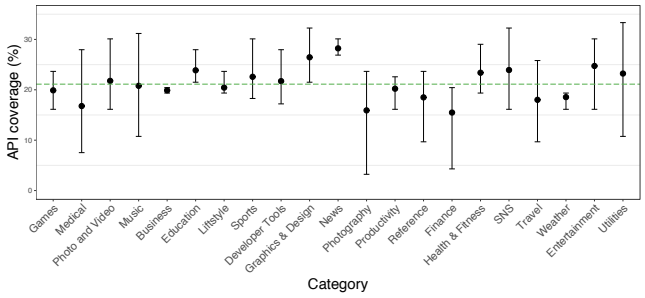


Figure 10: The coverage of API models derived by applications in each category. The error bars indicate the minimum and the maximum coverage of a program in each category.

summarized in Table 1. We note that 6 of the kernel panics came from the experiment in §5.3, where we fuzzed macOS with various different combinations of parameters. This result highlights the fact that mutation configuration is a critical factor in IMF.

The thread column of Table 1 shows in which thread a kernel panic has occurred. The first three kernel bugs are from a IMF thread, but all the other vulnerabilities are from a different thread. It is not surprising that kernel panics can happen in another thread, because IMF can corrupt the kernel state. One notable case is that of a `reboot` thread. Recall from §4.3.3, our watchdog process will reboot the OS when a kernel hang occurs. We found that sometimes the reboot process can cause a kernel panic.

The third column indicates where the crashing instruction belongs to. In most of the cases, crashing instruction was in the XNU kernel. The fourth column shows the list of kernel extensions appeared in the stack backtrace. Some kernel panics involve multiple

```
mov  rax, [rdi]
mov rsi, r14
call qword ptr [rax+0x50]
```

(a) Page fault.

```
mov  rdi, [rbx]
mov  rax, [rdi]
call qword ptr [rax+0x20]
```

(b) GPF (X).

Figure 11: Kernel panic cases.

kernel extensions, whereas some others do not have any related kernel extensions (denoted as '-' in the table). The fifth column, briefly describes the cause of each of the kernel panics. Most kernel panics were due to heap (zone) corruptions, GPF, NULL pointer dereference, and page fault. For GPF, we denote the cause within parentheses, e.g., "General protection fault (R)" means that GPF has occurred while reading a page.

Finally, the sixth column shows the impact of each of the kernel bugs. We manually investigated each of the kernel panics, and label them either with "DoS" or "likely exploitable". We specify whether a DoS is from a NULL dereference or not, because it may allow a control-flow hijack exploit under a certain circumstance (see §5.6.3). We say a kernel bug is likely exploitable if it satisfies one of the following conditions.

- The kernel crashed by dereferencing a pointer, but there is a following instruction within a basic block that jumps to an address referenced by the pointer.
- The kernel crashed with GPF while attempting to execute a non-executable page.

We now show two kernel panic cases that are likely exploitable, and one that is potentially exploitable in a certain environment.

5.6.1 Case 1: Page Fault. Figure 11a shows the 30th kernel panic listed in Table 1. The kernel crashed at the first `mov` instruction before the `call` instruction. A page fault was the cause: the `rdi` register was pointing to an unmapped address. However, we note that the following `call` instruction jumps to an address referenced by the `rax` register, which can be set by the `mov` instruction. Therefore, if we can allocate the memory at address `rdi` indicates or control `rdi`, we can control the instruction pointer. Thus, this is likely to be exploitable.

5.6.2 Case 2: GPF (X). The second case is taken from the second kernel panic in Table 1. In Figure 11b, the kernel crashed after the `call` instruction, and it threw a GPF because the address referenced by `rax + 0x20` was not an executable page. This means that either the `rax` register or the memory pointed by `rax + 0x20` is corrupted. Since at least one of them is already corrupted, it is highly likely that we can control the instruction pointer by corrupting the same point with a user-supplied value.

5.6.3 Case 3: NULL Pointer Dereference. The first kernel panic listed in Table 1 has a NULL pointer dereference. Traditionally, NULL pointer dereference kernel bugs are considered to be exploitable because an attacker could allocate a memory page at address zero using `mmap` or similar system calls [49] prior to triggering a null pointer dereference. However, latest XNU employs several defense mechanisms on such attacks: (1) disallowing memory allocation at address zero, and (2) leveraging a hardware-based protection mechanism, called Supervisor Mode Access Prevention (SMAP) [13], which prevents kernel code to access user-space memory. There

Idx	Thread	Crash Point	Kernel Extensions in Backtrace	Error Type	Impact
1	IMF	IOAcceleratorFamily2	IOAcceleratorFamily2	NULL pointer dereference	DoS (NULL)
2	IMF	Kernel	AppleIntelHD5000Graphics, IOAcceleratorFamily2	General protection fault (X)	Likely exploitable
3	IMF	Kernel	-	NULL pointer dereference	DoS (NULL)
4	ReportCrash	Kernel	AppleFSCompressionTypeZlib, HFS	VM map entry corruption	DoS
5	ReportCrash	Kernel	AppleFSCompressionTypeZlib, HFS	Zone VM object corruption	DoS
6	UserNotification	Kernel	AppleFSCompressionTypeZlib, HFS	VM map entry corruption	DoS
7	configd	Kernel	ApplicationFirewall	Zone cookie corruption	DoS
8	fontworker	Kernel	-	Zone cookie corruption	DoS
9	kernel_task	Kernel	BroadcomBluetoothHostControllerUSBTransport, IOBluetoothFamily, IOBluetoothHostControllerTransport, IOBluetoothHostControllerUSBTransport	Zone cookie corruption	DoS
10	kernel_task	Kernel	IOBluetoothHostControllerUSBTransport, IOUSBHostFamily	Zone cookie corruption	DoS
11	kernel_task	Kernel	BroadcomBluetoothHostControllerUSBTransport, IOBluetoothFamily, IOBluetoothHostControllerTransport, IOBluetoothHostControllerUSBTransport	Zone cookie corruption	DoS
12	kernel_task	Kernel	-	Zone cookie corruption	DoS
13	mds	Kernel	-	Release non-exclusive RW lock w/o refcount	DoS
14	mds_stores	HFS	HFS	Page fault	DoS
15	mds_stores	Kernel	AppleFSCompressionTypeZlib, HFS	Zone VM object corruption	DoS
16	mds_stores	Kernel	AppleFSCompressionTypeZlib, HFS	Zone VM object corruption	DoS
17	mdworker	HFS	HFS	Page fault	DoS
18	mdworker	HFS	HFS	Page fault	DoS
19	mdworker	HFS	HFS	Unlock not shared rw lock	DoS
20	mdworker	Kernel	HFS	General protection fault (R)	DoS
21	mdworker	Kernel	-	Release non-exclusive RW lock w/o refcount	DoS
22	reboot	Kernel	-	General protection fault (X)	Likely exploitable
23	reboot	Kernel	-	General protection fault (X)	Likely exploitable
24	reboot	Kernel	-	General protection fault (X)	Likely exploitable
25	reboot	Kernel	-	General protection fault (X)	Likely exploitable
26	reboot	Kernel	-	NULL pointer dereference	DoS (NULL)
27	reboot	Kernel	-	OSArray registry corruption	DoS
28	reboot	Kernel	-	OSArray registry corruption	DoS
29	reboot	Kernel	-	OSArray registry corruption	DoS
30	reboot	Kernel	-	Page fault	Likely exploitable
31	reboot	Kernel	-	Zone cookie corruption	DoS
32	reboot	Kernel	-	Zone cookie corruption	DoS

Table 1: Unique kernel panics found on macOS with IMF.

are still some possibilities to exploit NULL pointer bugs [51], but we conservatively mark those bugs as DoS.

6 DISCUSSION

This section discusses various issues related to API logging, API model inference, parameter mutation, and fuzzer engineering.

Logging APIs. The key intuition of our technique is that there are more than two API logs that have the common API call sequences with different parameter values. However, it is possible that a program itself manifests non-deterministic behaviors, even though the same input is used on the program. We believe even a non-deterministic application can have similar API logs if we collect an enough number of API logs. Furthermore, our technique is not application-specific. We can take any applications to gather legitimate API logs, and fuzz the OS that runs the program.

API Model Accuracy. The current value dependence inference of IMF only concerns about exact matching. However, there can be a value dependence between two values that are different from each other. Although we do not assume that we have source code, we can potentially figure out value dependences between a pair of output and input values by mutating the output value and observing the changes in the input value. One can also improve the ordering dependence inference by randomly removing function calls from a given sequence, or potentially get more exact value dependences with taint analysis. However, inferring more accurate API models is beyond the scope of this paper.

Mutation Strategy. Currently, IMF employs simple mutation strategies that perform XORing with a random value based on the parameters given by an analyst. Although it leverages parameter type information to handle several primitive types, it does not

handle high-level type information to mutate the parameter values. It is straightforward to adapt existing mutation strategies from other type-aware kernel fuzzers such as Trinity [25] and coverage-guided kernel fuzzer such as syzkaller [53]. Additionally, IMF can leverage value dependences obtained from the inferrer in order to dynamically generate API calls instead of replicating the same sequence in a loop. We believe these are potential future research directions.

Fuzzing Efficiency. IMF reboots the entire OS whenever it finds a kernel panic. This entails huge performance overhead, because as we found more bugs, we have to reboot the system more frequently. Typically, in our setting, it takes about 2-3 minutes to recover from a kernel panic, and resume the fuzzing campaign. We believe this problem can be overcome in several ways. One potential solution is to employ virtual machines. By using VMs, we can also run multiple instances of IMF on a single machine. We leave it as future work to incorporate VMs in IMF.

7 RELATED WORK

In this section, we survey related works in fuzzing, inferring models, and exploiting kernels. We refer the reader to §2 for kernel fuzzing.

Fuzzing. Although our main focus is on kernel fuzzing, there has been much research on userland fuzzing. Beyond simple mutation, there are the fuzzers that randomly generate inputs based on grammars, which are often called *generation-based fuzzers*, such as langfuzz [20] and Randoop [45]. A grammar in generation-based fuzzers is analogous to an API model in IMF. Unlike the existing generation-based fuzzers, however, we do not assume that such a grammar is given. Instead, we automatically infer it.

There are fuzzers that require an initial seed input to start generating test cases. Fuzzers in this category are often called *mutation-based fuzzers*, or simply mutational fuzzers. Recently many researchers have been focused on improving the mutational fuzzers [5, 8, 46, 47, 56]. We believe their mutation strategies can potentially be applied to kernel fuzzers including IMF.

API Model Inference. Mihai *et al.* [11] construct malware specifications by inferring a model from a syscall trace of a malicious program. However, their work is focused only on a single execution trace unlike IMF. There is a similar line of work by Choi *et al.* [10] who try to infer an API model, which they call a call-flow rule, from a userland API call trace. Their work share the common theme as Mihai *et al.*: they use only a single execution trace. Furthermore, their focus is on a combinatorial testing of API sequences, not on fuzzing the kernel API functions to find potential security vulnerabilities. There are automatic protocol reversing approaches [6, 15] from security community. There is also a whole field of research on automatic API specification inference in software engineering community. We refer to a recent survey on this field [48]. We note that our approach is complementary to all these techniques.

8 CONCLUSION

We have presented IMF, the first model-based kernel fuzzer that automatically infers API models from call traces. IMF produces multiple API logs from the same program with the same input in order to compute an API model. It then uses this model to generate a program that can automatically fuzz the OS under test. We have implemented IMF, and evaluated it on the latest macOS. As a result, we found 32 previously unknown kernel panics. Our experiments show that IMF has a practical impact on kernel security.

9 ACKNOWLEDGMENTS

We thank our shepherd, Brendan Dolan-Gavitt, and the anonymous reviewers for their helpful comments. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.B0717-16-0109, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis).

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. [n. d.]. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley.
[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the International Conference on Information Security*. 343–358.
[3] Paul Barton. 2013. PyUserInput. https://github.com/SavinaRoja/PyUserInput. (2013).
[4] Ian Beer. 2014. pwn4fun Spring 2014–Safari–Part II. http://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html. (2014).
[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1032–1043.
[6] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*. 317–329.
[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*. 209–224.
[8] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 725–741.
[9] Liang Chen, Marco Grassi, and Qidan He. 2016. Don't Trust Your Eye: Apple Graphics Is Compromised!. In *CanSecWest*. https://cansecwest.com/slides/2016/CSW2016_Chen-Grassi-He_Apple_Graphics_IS_Compromised.pdf
[10] YoungHan Choi, HyoungChun Kim, HyungGeun Oh, and Dohoon Lee. 2008. Call-Flow Aware API Fuzz Testing for Security of Windows Systems. In *Proceedings of the International Conference on Computational Sciences and Its Applications*. 19–25.
[11] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining Specifications of Malicious Behavior. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 5–14.
[12] CIFASIS. 2016. Neural Fuzzer. http://neural-fuzzer.org. (2016).
[13] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/. (2012).
[14] Jonathan Corbet and Greg Kroah-Hartman. 2016. Linux Kernel Development. http://go.linuxfoundation.org/linux-kernel-development-report-2016. (2016).
[15] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the ACM Conference on Computer and Communications Security*. 391–402.
[16] Michael Eddington. 2004. Peach Fuzzing Platform. http://peachfuzzer.com. (2004).
[17] Bernhard Garn and Dimitris E. Simos. 2014. Eris: A Tool for Combinatorial Testing of the Linux System Call Interface. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*. 58–67.
[18] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. *Security and Privacy in Communication Networks*. Springer International Publishing. 330–347 pages.
[19] Amaury Gauthier, Clement Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. 2011. Enhancing fuzzing technique for OKL4 syscalls testing. In *Proceedings of the International Conference on Availability, Reliability and Security*. 728–733.
[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the USENIX Security Symposium*. 445–458.
[21] George Hotz. 2013. machfuzzer. https://github.com/geohot/jenkyiphonetools/blob/master/machfuzzer. (2013).
[22] Apple Inc. 2013. Kernel Architecture Overview. https://developer.apple.com/library/content/documentation/Darwin/Conceptual/KernelProgramming/

Architecture/Architecture.html. (2013).
[23] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the ACM Conference on Computer and Communications Security*. 380–392.
[24] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the USENIX Security Symposium*.
[25] Dave Jones. 2010. trinity. https://github.com/kernelslacker/trinity. (2010).
[26] Mateusz Jurczyk. 2012. csrss_win32k_fuzzer. http://j00ru.vexillium.org/?p=1455. (2012).
[27] Rauli Kaksonen, Marko Laakso, and Ari Takanen. 2001. Software Security Assessment through Specification Mutations and Fault Injection. In *Communications and Multimedia Security*. 173–183.
[28] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In *Proceedings of the USENIX Security Symposium*. 957–972.
[29] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (2014), 2:1–2:70.
[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM Symposium on Operating System Principles*. 207–220.
[31] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. 1997. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the Symposium on Reliable Distributed Systems*. 72–79.
[32] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the Network and Distributed System Security Symposium*.
[33] MWR Labs. 2016. KernelFuzzer. https://github.com/mwrlabs/KernelFuzzer. (2016).
[34] Tin Le. 1991. tsys. http://groups.google.com/groups?q=syscall+crashme&hl=en&lr=&ie=UTF-8&selm=1991Sep20.232550.5013%40smsc.sony.com&rnum=1. (1991).
[35] Jonathan Levin. 2013. *Mac OS X and iOS Internals: To the Apple's Core*. Wrox.
[36] Moony Li. 2016. Active fuzzing as complementary for passive fuzzing. In *PacSec*.
[37] Lei Long. 2015. Optimized Fuzzing IOKIT in iOS. In *Black Hat USA*.
[38] MITRE. 2015. CVE-2015-5845. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5845. (2015).
[39] MITRE. 2015. CVE-2015-7077. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7077. (2015).
[40] NCC Group. 2016. Triforce Linux Syscall Fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer. (2016).
[41] Peter Oehlert. 2005. Violating Assumptions with Fuzzing. *IEEE Security and Privacy* 3, 2 (2005), 58–62.
[42] Dmytro Oleksiuk. 2009. IOCTL fuzzer. https://github.com/Cr4sh/ioctlfuzzer. (2009).
[43] Oracle. 2016. Kernel-Fuzzing. https://github.com/oracle/kernel-fuzzing. (2016).
[44] Tavis Ormandy. 2010. iknowthis. https://code.google.com/archive/p/iknowthis/. (2010).
[45] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering*. 75–84.
[46] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.
[47] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the USENIX Security Symposium*. 861–875.
[48] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637.
[49] sqrkkyu and twzi. 2007. Attacking the Core: Kernel Exploiting Notes. http://phrack.org/issues/64/6.html. (2007).
[50] Robert Swiecki and Felix Gröbert. 2010. honggfuzz. https://github.com/google/honggfuzz. (2010).
[51] Luca Todesco. 2015. Attacking the XNU Kernel in El Capitan. In *Black Hat EU*.
[52] Ilja van Sprundel. 2005. Fuzzing: Breaking software in an automated fashion. In *Chaos Communication Congress*.
[53] Dmitry Vyukov. 2015. Syzkaller. https://github.com/google/syzkaller. (2015).
[54] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*. 163–177.
[55] Vincent M. Weaver and Dave Jones. 2015. *perf_fuzzer: Targeted Fuzzing of the perf_event_open() System Call*. Technical Report. UMaine VMW Group.
[56] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 511–522.
[57] Chen Xiaobo and Xu Hao. 2012. Find Your Own iOS Kernel Bug. In *Power of Community*.
[58] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *Proceedings of the ACM Conference on Computer and Communications Security*. 414–425.
[59] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*. 183–192.
[60] Michal Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. (2014).
[61] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2011. Making Information Flow Explicit in HiStar. *Commun. ACM* 54, 11 (2011), 93–101.
[62] Markus Zimmermann. 2014. Tavor. https://github.com/zimmski/tavor. (2014).