KextFuzz: Fuzzing macOS Kernel EXTensions on Apple Silicon via Exploiting Mitigations

Tingting Yin, *Tsinghua University and Ant Group;* Zicong Gao, *State Key Laboratory of Mathematical Engineering and Advanced Computing;* Zhenghang Xiao, *Hunan University;* Zheyu Ma, *Tsinghua University;* Min Zheng, *Ant Group;* Chao Zhang, *Tsinghua University and Zhongguancun Laboratory*

https://www.usenix.org/conference/usenixsecurity23/presentation/yin

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

KextFuzz: Fuzzing macOS Kernel EXTensions on Apple Silicon via Exploiting Mitigations

Tingting Yin[1,3], *Zicong Gao*[4], *Zhenghang Xiao*[5], *Zheyu Ma*[1], *Min Zheng*[3], *Chao Zhang*[1,2*]

[1]*Tsinghua University* [2]*Zhongguancun Laboratory* [3]*Ant Group* [5]*Hunan University*
[4]*State Key Laboratory of Mathematical Engineering and Advanced Computing*

Abstract

macOS drivers, i.e., Kernel EXTensions (kext), are attractive attack targets for adversaries. However, automatically discovering vulnerabilities in kexts is extremely challenging because kexts are mostly closed-source, and the latest macOS running on customized Apple Silicon has limited tool-chain support. Most existing static analysis and dynamic testing solutions cannot be applied to the latest macOS. In this paper, we present the first smart fuzzing solution KextFuzz to detect bugs in the latest macOS kexts running on Apple Silicon. Unlike existing driver fuzzing solutions, KextFuzz does not require source code, execution traces, hypervisors, or hardware features (e.g., coverage tracing) and thus is universal and practical. We note that macOS has deployed many mitigations, including pointer authentication, code signature, and userspace kernel layer wrappers, to thwart potential attacks. These mitigations can provide extra knowledge and resources for us to enable kernel fuzzing. KextFuzz exploits these mitigation schemes to instrument the binary for coverage tracking, test privileged kext code that is guarded and infrequently accessed, and infer the type and semantic information of the kext interfaces. KextFuzz has found 48 unique kernel bugs in the macOS kexts and got five CVEs. Some bugs could cause severe consequences like non-recoverable denial-of-service or damages.

1 Introduction

macOS is one of the most widely used operating systems and has become a high-value and tempting attack target for adversaries. The drivers in macOS, which are also known as Kernel EXTensions (kext), introduce a large attack surface. There are hundreds of kexts loaded in macOS, accounting for half of the kernel code. Attackers can exploit vulnerabilities in the kexts to crash the system, bypass deployed security mechanisms, or even execute arbitrary code with kernel privileges. The vulnerabilities in macOS kexts can also affect other Apple-family operating systems such as iOS and iPadOS [2]. Therefore, discovering vulnerabilities in kexts is crucial.

Greybox fuzzing is one of the most effective ways to detect kernel (or driver) vulnerabilities, but they are not applicable to kexts in the latest macOS running on Apple Silicon. First of all, most of the existing greybox fuzzing solutions for kernels rely on source code, but kexts, in general, are closed-sourced. For instance, kernel fuzzers, including Moonshine [37], HFL [29], DIFUZE [23], and PeriScope [43] all need source code to extract knowledge (e.g., input formats of the interfaces) or monitor the execution (e.g., function hooking) of the targets under test. Kernel fuzzers that utilize the commonly used mechanisms KCOV [8] and KASAN [9] to track code coverage and catch runtime security violations also require source code too.

Second, existing fuzzing solutions for closed-source kernels also need various support from architectures or operating systems. For instance, kAFL [41] relies on a special CPU feature, i.e., Intel Processor Trace (Intel-PT), to collect the code coverage information. Digtool [38] leverages a customized virtualization layer to monitor the kernel execution based on Intel VT [6]. LLDBFuzzer [11] and SyzGen [21] intercept the kernel execution with kernel debuggers to collect coverage feedback and generate inputs. However, the latest macOS running on Apple Silicon does not have these supports, i.e., no CPU features like Intel-PT, poor virtualization support, and no active kernel debugging support [10].

Third, it is hard to infer interfaces of closed-source kexts. Both IMF [28] and SyzGen [21] use kext invocation traces to infer the interface dependencies. However, how to collect enough runtime driver invocation traces is still an open problem, so they can only deal with a small number of kexts. For instance, SyzGen can only collect useful traces for nine kext clients out of 25 clients they tested and only five of them contain useful information. Besides, IMF is only aware of the argument type of the top-level I/O Kit APIs with header files. SyzGen [21] uses dynamic symbolic execution to infer the argument types of interfaces and can cover primitive types like integer, string, and pointers, but cannot deal with complex

*Corresponding author: chaoz@tsinghua.edu.cn

input formats commonly used in kexts.

Lastly, macOS has deployed many mitigations to defeat potential attacks and make it difficult for dynamic testing. For instance, a majority of kexts use entitlement checks to guarantee that only userspace binaries signed by Apple or other specific developers could invoke their interfaces. According to our evaluation in Section 6.2, the ratio of such kexts could reach up to 61%, which is too large to ignore.

In this paper, we present the first smart fuzzing solution KextFuzz to *detect bugs in the latest macOS kexts running on Apple Silicon* and address the aforementioned challenges. Given that there are no source code and architecture support, KextFuzz tests real devices and utilizes binary rewriting to collect coverage information. However, it is not easy to rewrite kernel binaries in macOS because the kernel has a high requirement for stability. Analyzing the interface structure of closed-source targets is also challenging. Fortunately, we note that mitigations deployed in macOS also provide extra resources and knowledge of the kernel, which enables us to conduct fuzzing.

First, the latest macOS has used ARM Pointer Authentication Code (PAC) [31] to check the integrity of pointers (e.g., return addresses) at runtime and mitigate threats like control-flow hijackings. This mechanism will add instructions to initialize and verify pointers in basic blocks. However, these instructions can be neglected in the scenario of fuzzing. Firstly, we do not need to protect the system during fuzzing. Secondly, a pointer corrupted by random fuzzing is likely to trigger the system crash when it is dereferenced no matter whether it is protected by PAC. Therefore, we propose *a novel static binary rewriting scheme* by replacing ARM PA instructions in binaries. This scheme enables the fuzzer to alter or introspect the execution of the kexts, e.g., tracking code coverage. It does not break the structure of binaries (e.g., adding new code sections), and most importantly, does not import pointer dereference ambiguities, making the system running stable in kernel fuzzing. This method is also applicable to binaries with other extra instructions (e.g. Canary or Intel CET).

Second, a majority of the kexts perform entitlement checks during execution. Such mitigation could stop malicious userspace binaries from invoking privileged kext interfaces, and greatly improve system reliability, but meanwhile, it may cause blind spots to the system, as the protected kexts are not invoked and tested thoroughly. Therefore, we propose an automatic solution to *disable such entitlement checks* and test these kexts too. Specifically, it hooks and disables certain critical checker functions through static binary rewriting.

Third, macOS has provided userspace abstract layers (e.g., frameworks and daemons) for the kernel to avoid direct interaction between the user applications and the kernel. The abstract layer wraps complex kext invocations into well-defined functionalities. It leaves a chance for us to extract the knowledge of the kext interfaces from these wrappers, such as the type and values of the arguments. Therefore, we propose to analyze the binary code of kexts and their userspace libraries, to understand how arguments are used and generated and then infer the structures and semantics of the arguments. To better infer the semantics, we also propose *a taint analysis scheme based on the emulation execution of target binaries*. Compared to IMF [28] and SyzGen [21], this solution is more universal and practical.

We have implemented a prototype of KextFuzz based on Syzkaller [18] and evaluated it on macOS 11.5.2 with real Mac devices. With the novel static binary rewriting method, KextFuzz can track code coverage and find 6X more crashes than a black-box baseline fuzzing solution. KextFuzz can test all kexts that have entitlement checks and find bugs in them. The input templates generated by KextFuzz also leads to higher basic block coverage in fuzzing when compared with the state-of-the-art interface-aware kext fuzzer SyzGen. With the ability to collect coverage information, bypass entitlement checks, and generate high-quality testcases, KextFuzz finds 48 unique kernel bugs in a four-month fuzzing campaign, including out-of-bounds, use-after-free vulnerabilities.

In summary, we make the following contributions:

- We propose the first smart fuzzing solution KextFuzz to discover bugs in macOS kexts running on Apple Silicon, without relying on source code, execution traces, hypervisors, or hardware features.
- We leverage the mitigation instructions to present a novel static binary instrumentation scheme, which can alter and introspect kexts' execution. Specifically, it enables KextFuzz to track code coverage.
- We present a lightweight taint analysis solution based on emulation execution to infer interfaces of kexts by leveraging kernel isolation layers.
- We have developed a prototype of KextFuzz [1], deployed it on real devices, and found 48 unique kernel bugs.

2 Background

2.1 Kernel Extensions in macOS

MacOS kernel has two parts: the XNU kernel and kernel extensions (kexts). The XNU provides low level functionalities for the whole system, such as virtual memory management, low-level inter-process communication (IPC) mechanisms, and TCP/IP stack. Kexts extend the XNU to provide abstract kernel functionalities or interact with special hardware. Kexts can be viewed as drivers in Linux. Both of them are relatively independent kernel modules, but kexts are usually larger and more complex, which account for half of the kernel code.

Most of the kexts are developed based on the I/O Kit, an object-oriented macOS framework. In this framework, each kext may contain multiple services. Services are C++ classes

[1] https://github.com/vul337/KextFuzz

that ultimately inherit from the I/O Kit `IOService` class. They provide the core functionalities of a kext, such as life cycle control, device management, and interrupt control.

The services use user clients (subclasses inherit from `IOUserClient`) to manage the connection from the userspace processes. The clients provide interfaces that can be called from userspace via IPC (`mach_msg` system call) or I/O Kit userspace APIs. Each interface is numbered by a uint32 variable, which is also known as the function selector.

All the services in I/O Kit kernel extensions are managed by the I/O registry. The I/O registry is a dynamically updated database that records the currently registered I/O Kit objects and the relationship between them [7].

2.2 macOS Mitigations

Strict security mitigations in macOS make exploitation and dynamic testing difficult but also provide us with extra information and resources to find bugs automatically.

2.2.1 Pointer Authentication

ARM Pointer authentication (PA) is one of the advanced security mechanisms macOS adopts, introduced in the arm64e architecture. When running programs with Pointer authentication Code (PAC), pointer authentication would add a cryptographic signature to unused high-order bits of a pointer before storing the pointer while removing and authenticating the signature after reading the pointer from memory [14]. Any unexpected modification on the stored pointer value would invalidate the internal signature. When a signature is broken, the process would crash and no longer execute.

To enable PA, the compiler adds PA instructions before and after pointer access. Kexts on Apple Silicon-based macOS are compiled with these instructions. Kexts use PA to protect multiple types of pointers, including C++ V-Table, function return address, Objective-C method cache, computed goto labels, etc [12]. These PA instructions are distributed throughout the program, leaving spaces for us to modify the kext binary file. Not only the kexts, many other kinds of binaries also have instructions that can be replaced. The ELF binaries compiled with Canary or Intel CET instructions are examples.

2.2.2 Entitlement Checks

Entitlements are rights that can grant executable particular capabilities [1]. An application can have multiple entitlements at the same time, while different entitlements grant different capabilities. All the entitlements are embedded in executable binary file code signatures as key-value pairs. The kexts can check the entitlements to verify whether a userspace executable has permission to invoke the privileged code. In this way, entitlements restrict regular users from accessing the kexts. However, at the same time, the codes protected by entitlements are rarely called and not fully tested, making these codes popular attack targets.

2.2.3 Multi-Layered Operating System

To reduce direct interactive access from users to the kernel, many systems including macOS are designed to be a multi-layered operating system. From the lower to the upper layer, macOS includes the kernel layer, Core OS layer, Core Services layer, etc. The upper layers provide abstractions to the underlying kext behaviors. They include frameworks that wrap the complex kext interfaces into simpler APIs and daemons that can forward userspace requests to the kernel. This layered structure reduces the risk the kernel faces when dealing with non-standard user inputs because userspace processes usually do not have to interact with the kext directly.

The userspace abstract layers interact with the kernel in the most standard ways to ensure the stability of the system. They are updated and developed together with the XNU kernel and kexts. This leaves a possibility for us to learn the kext interface information from them.

3 Challenges

Due to the strict system security restrictions, special hardware environment, and closed development ecosystem, fuzzing macOS kexts on Apple Silicon faces additional challenges compared with other kernel fuzzing works. We classify the challenges into the following three aspects: lack of coverage feedback, strict runtime restriction, and complex input validation.

3.1 Coverage Feedback

Coverage feedback can guide the fuzzer to dig into untriggered code. One of the most common ways to collect coverage information is source-code level instrumentation. Fuzzers targeting binaries (e.g. kAFL [41] and AFL-qemu [5]) can trace the program with hardware mechanisms or virtualization. Developers can also set breakpoints on basic blocks and collect coverage in the debugger as lldb-fuzzer [11] and SyzGen [21].

However, these methods are not available on the latest macOS. Apple Silicon does not provide hardware monitors like Intel-PT. Virtualization support is also limited due to the unique hardware devices and customized instruction set. Even if the system can boot in a virtual machine, many kexts can not work [17]. According to our analysis, even in x86_64 version macOS which has mature virtualization support, there are still half of the kext services not available in the virtual machine [2]. Active debugging is also not supported either as declared in the macOS kernel debugging kit [10]. Not only macOS, operating systems running on other dedicated hardware, like embedded systems, are facing a similar situation.

[2] macOS 11, VM Fusion virtual machine and Intel MacBook pro 16 inches

Without hardware and sound virtualization supports, static binary rewriting is the most efficient way to collect the coverage information. However, existing methods have fundamental limitations when applied to macOS kexts. Firstly, kexts have high requirements for stability because they work in kernel mode. Any errors in kexts will let the system crash and reboot, which is time-wasting and limits the fuzzer to test deep code. However, as discussed by STOCHFUZZ [47] and RetroWrite [25], existing static rewriting methods generally introduce reference ambiguities [4, 25], which causes crashes easily. Solutions based on IR lifters [33, 35] are also facing similar challenges in identifying and recovering references. Secondly, some solutions are not practical when applied to the kernel, especially the macOS kernel. STOCHFUZZ [47] works through trial and error. It needs to crash and re-instrument the targets many times before determining the final rewriting solution, which means rebooting the system and re-install the kernel repeatedly in the kernel fuzzing scenario. e9patch [26] only supports x86_64 binaries and uses a complex virtual memory layout. In summary, we need a rewriting solution that works stably, does not need complex toolchain support, and is easy to be implemented.

3.2 Strict Runtime Restriction

Runtime restrictions in kexts are hardly discussed in the previous work. Kexts in macOS only allow specific userspace executables to invoke privileged code via "Entitlement" checking. The entitlements are rights or privileges that grant an executable particular capabilities [1]. Executables store entitlements as key-value pairs embedded in their code signatures.

Most of the Entitlements are private or semi-private. Only the executables developed by Apple and a few specific companies [24] can use them. Programs like fuzzers developed by third parties can not access code protected by entitlements without spoofing signature checking. The method of spoofing entitlements checks varies with the macOS versions, and the entitlement value meets the requirement of kexts that need to be found via reverse engineering.

However, the kext protected by private entitlements are still attractive attack surfaces. Attackers can exploit them by building exploit chains. A common method is to compromise a system process with the entitlements to call the privileged code [15]. However, building an exploit chain is non-trivial manual work. To detect bugs nested in these kexts, the fuzzer needs to bypass the entitlement check automatically.

3.3 Complex Input Format

The kext interface structures can be complex. Listing 1 is a simplified example of vulnerable kext interfaces and their bug POC. To trigger the vulnerability in `set_device`, the fuzzer needs to call `create_device` first to get a valid device `id`. The `create_device` function takes in an XML string

```
1  // kext interfaces
2  int create_device(void * input, void * output) {
3      v0 = OSUnserializeXML(input, ...);
4      v2 = TypeCast(v0,OSDictionary::metaClass);
5      if(!v2) { return ERROR; }
6      device = gen_device(v2);
7      if(!device) { return ERROR; }
8      // write the device id to the output
9      *((int *)output) = device->id;
10     return 0;
11 }
12 int set_device(void * input, void * output) {
13     // read the device id from input
14     id = ((int *)input)[0];
15     v0 = get_device(id);
16     if(!v0) { return ERROR; }
17     // vulnerable_code();
18     return 0;
19 }
20 // POC
21 input = IOCFSerialize(cfdict, 0);
22 //input="<dict>
23 //       <key>IOSurfaceClass</key><string>XX</string>
24 //       <key>IOSurfaceHeight</key><int>0x40</int>
25 //       </dict>";
26 create_device(input, output);
27 id = ((int *)output)[0];
28 set_device(&id, output);
```

Listing 1: vulnerable kext interfaces and the POC.

variable `input` and deserializes it to create a dictionary object following the Apple CoreFoundation development standard. The CoreFoundation is a wildly used framework in macOS, which uses serialized objects to pass data among programs. A valid `input` example is shown in the list. If the `input` provides proper data in the correct format, `create_device` will generate a device object and write its `id` to the `output`. Then, the attacker or the fuzzer can call `set_device` with a valid `id` to trigger the bug.

This example shows that an effective testcase should meet interface requirements in two aspects: value and type. In terms of value, KextFuzz needs to assign arguments like `id` according to the context. In terms of type, it not only needs to deal with primitive types like pointers or integers but also complex types like CoreFoundation serialized object expressions.

However, kext binaries provide little interface information and are hard to be analyzed. Firstly, kexts do not explicitly declare the value dependencies (e.g., `id`) between interfaces. All the interfaces are exported equally and independently. Secondly, kexts declare all arguments as `void*` pointers with no type information and parse them on need. The intuitive approach that identifies the argument type by analyzing their parsing process may need complex inter-procedural analysis. For example, the `input` argument is deserialized in the `create_device` but parsed in `gen_device`. We need to analyze the `gen_device` to know the expected elements of `input`. In real cases, parsing processes can be nested deeper.

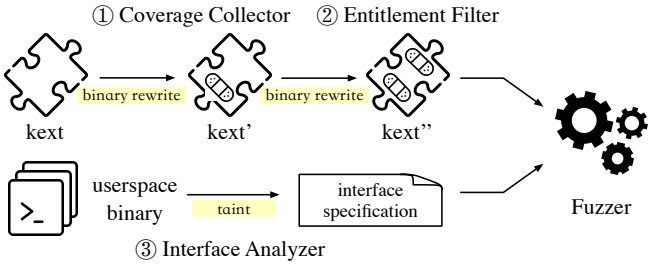

Figure 1: Workflow of KextFuzz.

4 Design

To address the aforementioned challenges, KextFuzz designs three components: the coverage collector, the entitlement filter, and the interface analyzer. Rather than succumbing to the strict system mitigations and using compromising testing solutions, KextFuzz exploits mitigations to enable smart fuzz. At a high level, it leverages resources reserved for mitigations to introspect the kext execution and extracts additional knowledge from mitigations to guide fuzzing.

The workflow of KextFuzz is shown in Figure 1. KextFuzz preprocesses the kexts before fuzzing them through the coverage collector and the entitlement filter. Then, it uses the interface analyzer to extract interface specifications from userspace libraries and daemons calling the kext interfaces.

4.1 Coverage Collector

Coverage feedback is essential for greybox fuzzing. It helps the fuzzer find high-quality seeds that can reach deeper code paths. However, as introduced in Section 3.1, common ways of collecting coverage like source code level instrumentation, hardware tracers, and traditional binary rewriting solutions can not be applied to macOS kexts.

KextFuzz implements a novel binary-level instrumentation solution to solve this challenge via instruction substitution. It leverages the code space used by PA instructions and replaces them with coverage collection instructions. In this way, KextFuzz can add instructions to collect coverage without importing pointer dereference ambiguity or breaking Mach-O file structure. As a result, it works stably and is available for all of the kexts. The PA instructions are designed to guard kexts against malicious pointer modifications in real-world attacks. However, they can be neglected in the fuzzing scenario. Firstly, we do not need to protect the system from attacks during fuzzing. Secondly, the pointer corrupted by the random fuzzing inputs can usually crash the system itself without PA protection. Therefore, KextFuzz removes the PA instructions in kext binaries and instruments new instructions into the leaving interspace.

Figure 2 is an example of the instruction substitution. In this case, the kext uses PA instructions to authenticate the vtable address stored in `[x0]`. The kext authenticates the address in lines 1-6 and checks the result in line 7. If the check

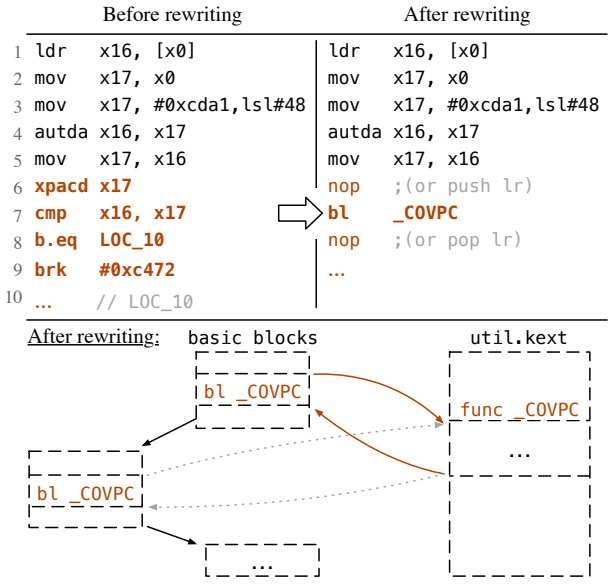

Figure 2: KextFuzz replaces PAC instructions to call coverage collection function `_COVPC`.

is passed, the control flow continues to line 10. Otherwise, the program considers the vtable address has been modified by an attacker and breaks the execution at line 9. With this authentication, the system can protect the kernel from a control flow hijacking attack. KextFuzz replaces the PA instructions to call the coverage collector function `_COVPC` in lines 6 to 9 via binary rewriting. After rewriting, the program will call `_COVPC` function to record the current basic block address and then return to the original program.

KextFuzz implements `_COVPC` function in an independent kext. The instrumented `bl` instruction (line 7) records the caller's PC in the link register `lr`. `_COVPC` function records the value of the `lr` register as coverage information. To avoid destroying the register context, KextFuzz records the original `lr` register on the stack if it is not stored yet. `_COVPC` further records the other registers at its function entry and recovers them before returning to the original control flow.

KextFuzz can instrument kexts at basic block granularity roughly because the kexts are developed in C++ and widely use PA instructions to protect return addresses and indirect calls. In addition, the PA instructions distribute at different points of the program. By sampling the signal of these instrumented points, the fuzzer can approximately know the depth and breadth of the current triggered code.

4.2 Entitlement Filter

KextFuzz exploits the entitlement checker to test privileged code. Entitlements protect sensitive functionalities from being called by normal users but increase the difficulty of dynamic testing, making these codes lack tests. KextFuzz bypasses these checks like the attackers but does it automatically.

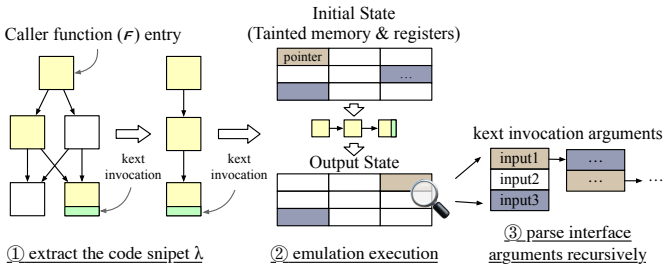

Figure 3: Interface structure identification process

Kexts invoke the checker functions to perform entitlement checks. These functions are implemented by macOS XNU and AMFI (AppleMobileFileIntegrity) kext. Kexts that need entitlement checking have to call the checker functions as external functions.

KextFuzz hooks the checker functions via binary rewriting to hijack entitlement checks. Firstly, KextFuzz exports a fake checker function in a self-developed kext, which always returns true for entitlement checks. Then, it rewrites the external function symbol table of the target kext and replaces the symbol of the actual checker function with the fake one. The fake checker function symbol will cheat the kernel binary linker (kernel management utility, kmutil) to link the fake checker to the kext. Kmutil links all the kexts as a kernel collection before loading it into the system. During the link process, the kmutil creates trampolines for external functions and redirects the function invocations to the trampoline (function stub), including the entitlement checker function. The function symbol is an identifier between the invocation and the trampoline. If the kext uses a fake checker function symbol, kmutil will redirect the entitlement check invocation to the trampoline of the fake checker. More details about rewriting the function symbol can be found in Section 5.2.

The whole process can be done automatically. Compared with the solutions of adding entitlement to the fuzzer, KextFuzz does not have to recognize the required entitlement via complex static analysis. With this method, KextFuzz can completely remove the entitlement checks from the kext and test the privileged functions sufficiently.

4.3 Interface Analyzer

We notice that the kext userspace "wrappers" can provide abundant interface information. To mitigate the potential risk from non-standard user input, macOS provides abstract layers for kernel services in userspace, which includes frameworks, libraries, system daemons, etc. These components encapsulate complex kext invocations into well-defined services and interact with kexts in standard ways. They call kext interfaces in proper sequences and set arguments that meet the value and type requirements. By analyzing these wrappers, KextFuzz can infer the interface structures. However, the wrappers are also closed-source, so we have to find a binary-only method to extract interface information automatically.

KextFuzz designs a lightweight multi-tag static taint method to analyze kext userspace wrappers. The overall process of the taint analysis is shown in Figure 3. KextFuzz only analyzes the kext invocation-related code snippets to save time. It defines a group of taint sources that may provide interface information, such as the output of kext invocations and stack pointers. It extracts the related code snippets from the wrappers, runs them in an emulator, and adds the taint tags on registers and memory during the execution. The taint tags will spread into the kext interface arguments after the execution. Therefore, KextFuzz can infer the argument attributes by analyzing the taint tags.

More specifically, to get the kext invocation code snippets from wrappers, KextFuzz builds control flow graphs (CFG) of the functions ($\mathcal{F}$) which invoke the kext interfaces (I). Then, KextFuzz extracts the paths (λ) starting with $\mathcal{F}$'s function entry and ending with I as analysis targets. The code in these paths prepares and initializes the kext invocation arguments, so KextFuzz can learn the argument information from them. This process can be extended recursively to conduct interprocedural analysis. Given a function $\mathcal{F}'$ which invokes $\mathcal{F}$, we could append λ with a prefix path, which starts from $\mathcal{F}'$ entry point to its invocation site of $\mathcal{F}$. Note that, the λ always starts with a function entry and end with a kext invocation.

Then, KextFuzz performs taint analysis on λ. It defines the following types of taint sources:

- Kext interface output (*s1*). Some kext interfaces use the outputs of other interfaces as inputs. We refer to these outputs and inputs as resource variables. Their values are context-sensitive, which is hard to generate via mutation. This tag can help KextFuzz to recognize them.
- Global variables (*s2*). Many developers store the values of resource variables in global variables, so we also add tags on global variables to identify resource variables heuristically.
- Object creation function return values (*s3*). As discussed in Section 3.3, kexts use CoreFoundation style inputs, which have complex formats. These variables are created by specific APIs, e.g., function `CFArrayCreate` is used for creating CFArray objects. We list these APIs manually and add tags on their return values if they are invoked in λ.
- Stack and heap pointers (*s4*). KextFuzz adds taint tags on stack frame registers (`SP`, `X29` in arm64) and return values of memory allocation functions (e.g. `malloc`) to recognize pointer type arguments.
- Caller function arguments (*s5*). Binaries developed in C++ export external methods with mangled function symbols, which have type information of the arguments. Therefore, KextFuzz adds tags on the arguments of the caller function $\mathcal{F}$. If the I uses the arguments of $\mathcal{F}$, KextFuzz can get their type from the function symbols or decompilers.

KextFuzz adds taint tags on taint sources before and during

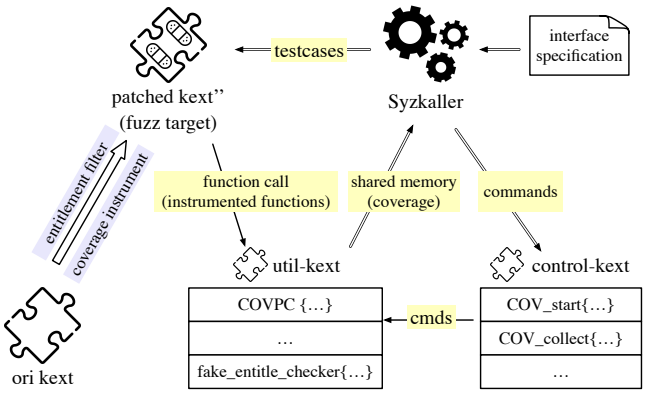

Figure 4: KextFuzz architecture

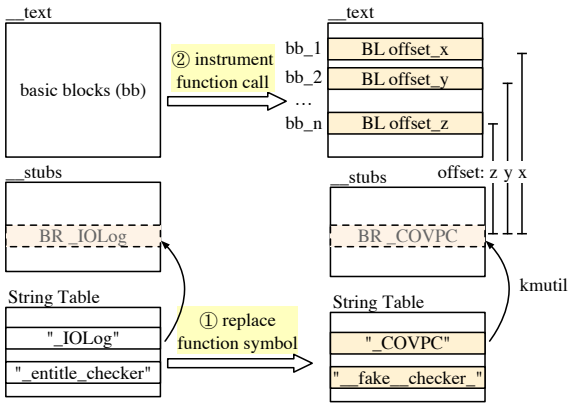

Figure 5: Instrument a kext via static binary rewrite

executing λ. After execution, some tags may spread into kext invocation arguments. KextFuzz parses the tags in the argument registers to infer their format. One contiguous memory block with the same taint tag will be grouped into one field. If the argument is tainted as a pointer, KextFuzz will further parse the memory it points to. Thus, KextFuzz can recognize the structures and the pointers nested in structures. If the input field has no tag, we suppose it is an `int8` type array and feeds them with random data during fuzzing. The implementation of the taint system is further introduced in Section 5.3.

5 Implementation

5.1 Architecture

Figure 4 shows the overall workflow of KextFuzz. KextFuzz patches the target kext via static binary rewriting before fuzzing it. Compared with the original kext, the patched kext (the `kext″` in Figure 4) is instrumented with the coverage collector function calls and has no entitlement restrictions.

KextFuzz uses two kexts (`util-kext` and `control-kext`) to facilitate the greybox fuzzing. The `util-kext` exports the coverage collector function and the fake entitlement checker function. The fake entitlement checker ensures the fuzzer can pass the entitlement check to access the privileged code. The coverage collector function records its caller's PC as coverage information. It only records the coverage of the fuzzer thread to avoid noises. `control-kext` provides KCOV-like interfaces. Thus, the system can be easily adapted to various fuzzing frontend. We use Syzkaller in our implementation.

5.2 Static Binary Rewrite

KextFuzz patches kexts via binary rewriting to collect coverage information and hooks entitlement checks. The instrumentation process is simple and efficient because the lightweight instrumentation method proposed by KextFuzz does not break any references, increase binary size, nor need a special memory layout.

As shown in Figure 5, KextFuzz uses two steps during instrumentation. First, it replaces an existing function symbol to `_COVPC` to reuse its stub entry. Then, KextFuzz replaces the PA instructions with the instructions (`BL offset`) that call the `_COVPC` stub. The `BL` instruction records the caller's PC in the link register `lr`. The `_COVPC` function records the value of the `lr` register to an internal buffer as the coverage information and maps it to shared memory when needed.

The `_stub` is a Mach-O file section that works as a trampoline to support external function calls. When an executable needs to use an external method, it calls the corresponding stub entry first. The stub will redirect the control flow to the real function address, which is filled during the link stage. For kexts, they call functions in other kexts as external methods. KextFuzz reuses the existing function stub entries for `_COVPC` function rather than adding a new one. Adding new stub entries requires correcting the section size and relative references according to the file structure, which changes while the system updates. Reusing existing ones makes the implementation simplifier and improves compatibility. The replaced stub entry should belong to a function that has no actual impact on the data flow and control flow of kext execution. Logging functions (`IOLog`, `_os_log_internal`, etc.) are good choices. KextFuzz successfully instrumented all the kexts we've tested so far without observable side effects by replacing these two logging functions.

The process of hooking the entitlement checker function is similar, but KextFuzz only needs to perform the first step, i.e., replace the symbol of the checker function with the symbol of the fake checker function.

5.3 Taint via simulation execution

KextFuzz designs a lightweight taint method based on emulation execution to analyze code snippets λ extracted in Section 4.3. We use Triton [19] as the emulation execution engine. KextFuzz has to overcome two challenges in this task. Firstly, since KextFuzz only analyzes code snippets rather than the whole program, how should it initialize registers and memory

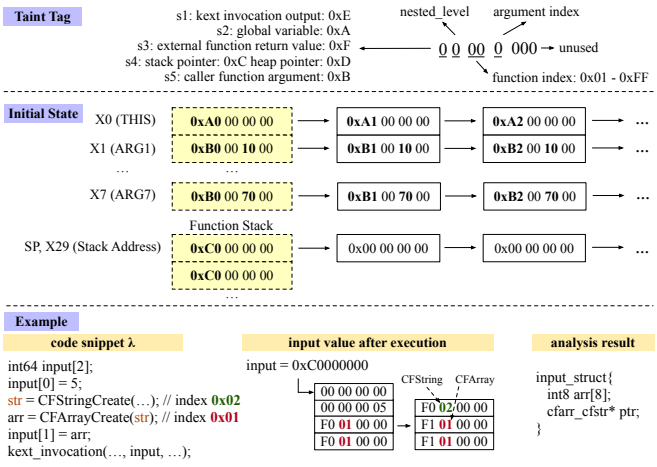

Figure 6: The taint tag encoding format, the initial state, and a parsing example of KextFuzz Interface Analyzer.

before execution? Secondly, λ may call other functions during execution. How should we emulate the behavior of these functions, especially the functions creating CoreFoundation objects which provide argument type information?

To answer the first question, KextFuzz uses special values to initialize the program state and uses these values as taint tags. Since λ always starts with function entries, KextFuzz has to initialize the function arguments and function stack pointers before execution, which corresponds to taint source *s4* and *s5*. Special values used in initialization are encoded as shown in Figure 6. KextFuzz groups the registers and memory into 32-bit units. In each unit, KextFuzz uses the high bits to record the taint source and detailed information (e.g. function and argument index). It also records the nested level to recognize pointers. The nested level refers to the dereference times needed for accessing the value. Pointers use level zero. The memory chunks they point to use level one, and further, level two and three. Before execution, KextFuzz initializes the registers and memory as shown in Figure 6. The arguments as well as the memory they point to are initialized with *s5* tags. Therefore, taint tags will not be lost during pointer dereferences. The wrapper binaries in macOS are mainly developed in C++, so KextFuzz supposes the first argument of $\mathcal{F}$ stored in `X0` is a `THIS` variable pointer, which can be viewed as a global variable and needs the tag *s2*.

To answer the second question, KextFuzz creates a generic function abstraction $\mathcal{M}$ for CoreFoundation object creation functions. $\mathcal{M}$ follows a common pattern of these functions, i.e., takes the value and size as inputs and returns a pointer pointing to a memory chunk. It adds `s3` tag on the returned pointer and the memory chunk. The taint tags of the arguments are recorded as the head of the memory chunk, in case the object creation process is chained, e.g., creating a string object first and then using it as an argument to create an array object.

In the example shown in Figure 6, the `input` of the kext

```
match = IOServiceMatching("service_name");
IOServiceGetMatchingServices(0, match, &iterator);
service = IOIteratorNext(iterator);
IOServiceOpen(service, ..., client_type, &conn);
IOConnectCallMethod(conn, func_sel1, in_ptr1, out_ptr1, ...);
IOConnectCallMethod(conn, func_sel2., in_ptr2, out_ptr2, ...);
```

Figure 7: An example of KextFuzz testcases.

invocation is an `int64` array stored on the stack. λ assigns a scalar variable 5 to `input[0]` and a CoreFoundation array's pointer to `input[1]`. After the execution, the input is tainted with a stack pointer tag 0xC. The value stored in address 0xC0000000 is a variable with no tag and a pointer with the *s3* tag, which means it is returned by `CFArrayCreate`. This pointer points to a memory chunk with another *s3* tag of `CFStringCreate`. Therefore, KextFuzz knows that the `input` is a pointer of a struct that has a scalar variable and a CoreFoundation array containing string-type items.

Using special values as taint tags makes KextFuzz taint analysis lightweight. The taint tags are stored as registers and memory values, so they can be spread automatically while executing the code. Therefore, KextFuzz does not have to write complex taint propagation rules. Secondly, these values distinguish the heap space, stack space, and registers, making the initialization state close to the real program state, so KextFuzz can only execute the code snippets rather than analyze the whole program.

5.4 Test Cases Generation

As introduced in Section 2.1, a fuzzer has to get a kext client connection before invoking their interfaces. An effective testcase is shown in Figure 7. In addition to generating inputs according to the interface structures recognized in Section 4.3, KextFuzz also needs a valid `service_name` and a `client_type` to get the connection, and a function selector `func_sel` corresponding to the targeted interface index.

KextFuzz gets valid values of these inputs by analyzing the kext binaries statically. The service name is a string that equals the class name of the kext service class. This type of class has to inherit specific functions from their base class `IOService`, so that KextFuzz can identify them according to function symbols. The service class rewrites the `newUserClient` function of `IOService` to create clients according to the `int32` type input `client_type`. KextFuzz builds a CFG for the rewritten function and analyzes the compare expressions on `client_type` argument to know its candidate value. The client class implements a dispatch function to get the interface function from a function table according to the index variable `func_sel`. The function table also stores the size of each parameter. KextFuzz gets the function table by analyzing the objects that the dispatch function refers to. For the classes

that do not rewrite the above functions, KextFuzz further gets inheritance information from the I/O registry [7] introduced in Section 2.1 and uses their parent classes information for them. With these information, KextFuzz can get the client connection before calling the kext interfaces.

For the interface arguments, KextFuzz generates and mutates them according to the interface formats. For string type arguments, it uses the strings in the wrapper binaries as seeds. For resource variables, KextFuzz uses the outputs of other interfaces as their seeds.

5.5 Crash Data Collection

KextFuzz runs fuzz on real macOS devices. The system can record the coredump and attempt to reboot after a post-panic crash dump when setting the boot-arg debug variable to DB_KERN_DUMP_ON_PANIC | DB_REBOOT_POST_CORE. When the crash happens, developers can analyze crashes according to the backtrace and then analyze the core dump manually to find the root causes.

6 Evaluation

We conducted experiments to answer the following questions:

- RQ1: How many basic blocks can the Coverage Collector instrument? How much overhead does it cost? Can it help KextFuzz find more bugs?
- RQ2: How many kexts have entitlement checks? Can Entitlement Filter help KextFuzz bypass these checks?
- RQ3: How many valid services and clients can KextFuzz identify?
- RQ4: How does the interface specification generated by KextFuzz compare with the one from SyzGen? How much do the different components of KextFuzz contribute to fuzz?
- RQ5: How many bugs can KextFuzz find in kexts?

Experiment Setup. We conduct our experiment on four machines: two MacBook Pro and two Mac Mini devices with Apple M1 chip. For each group of comparison experiments, we run them on the same device. The macOS version we tested is 11.5.2. To avoid unrecoverable damage to the device, we remove risky-kexts from each experiment dataset, like disk-related kexts, low-level hardware kexts, etc.

6.1 Evaluation of Coverage Collector (RQ1)

KextFuzz uses a novel strategy to instrument the kexts at the binary level without breaking the offset between original instructions or destroying the Mach-O file structure. In this section, we test KextFuzz on different kinds of kexts shown in Table 1 to evaluate the effectiveness and efficiency of the Coverage Collector. The tested kexts are all fundamental or frequently used kext of different system functionalities.

Table 1: The instrument basic block rate and the overhead of the Coverage Collector

kext	instrumented	cov-aware	overhead
IOSurface	26.86%	32.09%	3.23x
IOMobileGraphicsFamily-DCP	24.09%	30.10%	3.74x
AppleH13CameraInterface	35.81%	38.63%	4.74x
AUC	28.36%	35.45%	3.76x
IONetworkingFamily	31.88%	37.35%	1.40x
AppleBCMWLANCore	16.19%	18.98%	1.02x
AppleIPAppender	33.80%	41.59%	2.29x
IOUSBHostFamily	33.20%	35.88%	2.24x
IOUSBDeviceFamily	32.70%	37.62%	2.57x
IOAudioFamily	37.81%	41.65%	1.17x
IOAVBFamily	75.26%	78.95%	-
AppleAOPVoiceTrigger	49.91%	55.22%	0.96x
AppleMultitouchDriver	37.74%	41.98%	2.78x
IOHIDFamily	34.84%	39.42%	1.37x
EndpointSecurity	18.44%	25.44%	1.07x
AppleBluetoothDebug	38.80%	43.82%	0.85x
AppleBluetoothModule	22.66%	28.05%	0.97x
IOBluetoothFamily	31.89%	34.99%	0.76x
IOReportFamily	49.23%	51.69%	1.62x
Average	34.71%	39.42%	2.03x

Effectiveness. To evaluate the effectiveness of KextFuzz, we count the instrumented basic block rate and compare KextFuzz with a black-box fuzzing baseline in terms of bug detection. Firstly, we count the proportion of the instrumented basic blocks and coverage-aware basic blocks in the instrumented kexts. Coverage-aware basic blocks include the instrumented basic blocks, the full dominator that all of its successor nodes are instrumented (or all its successors have an instrumented post dominator), and the full post dominator that all of its predecessor nodes are instrumented (or all its predecessors have an instrumented dominator). As shown in Table 1, KextFuzz can instrument about 34.71% of basic blocks and make 39.42% of basic blocks coverage-aware. Functions like _IOLog redirected to _COVPC could collect coverage too, which enables instrumentation 4.15% of blocks.

This instrumented rate can still be improved with engineering efforts but has already shown great effectiveness in bug detection. To verify the ability of the coverage collector to help the fuzzer discover bugs, we run KextFuzz for 24 hours with and without the coverage collector in the same environment. KextFuzz finds six unique bugs with coverage feedback. However, without coverage feedback, KextFuzz can only find one bug. We repeated this evaluation three times. KextFuzz with coverage collector finds six or seven bugs in each round, while KextFuzz without coverage feedback can only find one.

Efficiency. We run KextFuzz on kexts with and without coverage instrumentation to evaluate the overhead introduced by the coverage collector. We fuzz each kext for one hour and record the throughput (i.e., testcases executed during fuzzing). As shown in Table 1, the average overhead is 2.03x. One kext does not have overhead data because KextFuzz detected bugs in it during the testing. The bug make the device reboot frequently, which is time-consuming and makes it difficult to calculate the real throughput. We also find that some kexts

run slightly faster with the coverage instrumentation. This is probably because KextFuzz hooks the logger functions, saving the time of logging during fuzzing, and these kexts invoke logging functions frequently.

Stability. We have run KextFuzz on four real macOS devices for months, and the system has not crashed due to the coverage instrumentation yet.

In summary, the coverage collector of KextFuzz can collect adequate coverage feedback with great efficiency. Based on the insight of KextFuzz, The instrumented rate can be further improved with extra engineering effort because we only replace two kinds of PA instruction in our prototype.

6.2 Effectiveness of Entitlement Filter (RQ2)

As introduced in Section 4.2, some kexts use entitlement checks to restrict the userspace processes from accessing privileged code. KextFuzz hooks the entitlement check via static binary rewrite to test privileged kexts.

To evaluate the effectiveness of the entitlement filter, we first count the kexts with entitlement check functions by analyzing their external function symbols. There are 229 kexts loaded on our device (Mac mini, M1 2020). 93 of them have clients that can be connected from the userspace. Among the kexts that have clients, 57 kexts have entitlement checks.

Entitlement Filter enables KextFuzz to find bugs in less tested and privileged kexts. Among the 57 kexts that have entitlement checks, eight of them are security-related (e.g., `BootPolicy`, `EndpointSecurity`), nine of them are system management related (e.g., system tracing framework, hardware monitor, and controller), and the others of them are the fundamental frameworks for key services like disk, graphic, network, firmware, etc. These kexts work as the cornerstone of the operating system but are hardly being tested because of the entitlement checks.

KextFuzz bypasses the entitlement checks successfully with the entitlement filter and finds 18 more bugs with it. The entitlement filter expands the range of the code that KextFuzz can test. It hooks the entitlement check statically and does not have extra runtime overhead. As for stability, we do not find negative effects introduced by the entitlement filter during the long-time fuzzing.

6.3 Effectiveness of Interface Analyzer (RQ3)

To demonstrate the effectiveness of KextFuzz in identifying interfaces, we evaluate the number of valid services, clients, and functions found by KextFuzz in kexts and compare it with the state-of-the-art fuzzing tool SyzGen [21].

We run KextFuzz on Mac Mini (M1, 2020). SyzGen can only support Intel-based macOS, so we run it with the same environment setting in its paper: a VMFusion virtual machine on an Intel MacBook Pro device. The operating system is macOS 11.5.2 with both ARM and x86_64 versions. Part of the kexts in this version are cross-compiled with the same source code. To avoid the differences caused by the environment, we filter out the common services that exist in both environment for comparison. A service or client is considered valid if it can be matched or connected to the system.

The result shows that KextFuzz finds 70 valid services and 97 clients, while SyzGen finds 43 services and 43 clients in total. Among the services existing in both environments, KextFuzz finds all the services and clients that are found by SyzGen. We evaluate the quality of interface specifications found by KextFuzz and SyzGen during fuzzing in Section 6.4.

6.4 Fuzzing with KextFuzz (RQ4)

Experiment Setup. We run KextFuzz with different configurations to figure out the contribution of each component and compare the interface specifications generated by KextFuzz and SyzGen. We annotate the configurations as follows:

- **KF-K.** Fuzz kexts with interface information extracted from the kext binaries only (§5.4). The interface information includes service name, client type, function selector, and the argument sizes.
- **KF-En-K.** Fuzz kexts patched by the entitlement filter with interface information extracted from the kext binaries. Compared with KF-K, KF-En-K can test privileged codes protected by entitlement checks.
- **KF-En-K&U.** Fuzz kexts patched by the entitlement filter with interface information extracted from both userspace binaries and kexts. In this configuration, the fuzzer can generate inputs according to the grammar and semantic information from kext userspace libraries or wrappers, so KextFuzz can recognize arguments with resource variables and complex structures (e.g., CoreFoundation arguments).
- **KF-En-SyzGen.** Fuzz kexts patched by entitlement filter with templates generated by running SyzGen's concolic symbolic execution engine. In this configuration, we can compare the quality of interface specifications generated by KextFuzz and SyzGen.

We run SyzGen to generate templates in the same environments mentioned in Section 6.3. To make the experiment fair, we manually find the kexts cross-compiled from the same source code as the fuzzing targets. These kexts have the same interfaces in two architectures. Therefore, we can run SyzGen on x86_64 macOS to generate the templates and use them in fuzzing. We conducted the experiment on the kexts clients both KextFuzz and SyzGen identified in Section 6.3 and removed the risky clients (e.g., disk-related) from them to avoid causing damage to the system during the long time evaluation. As a result, the remaining targets can be grouped into five classes: USB-related, Graphic-related, Audio-related, HID-related, and Network-related, as shown in Table 2. We run each configuration for 24 hours on each group of kexts and repeat the experiment three times.

Table 2: Fuzzing targets in Section 6.4

type	clients
Audio	IOAudioEngineUserClient IOAVBNubUserClient
Network	IOUserEthernetResourceUserClient IONetworkUserClient
Graphic	IOSurfaceRootUserClient
USB	AppleUSBHostInterfaceUserClient AppleUSBHostFrameworkInterfaceClient AppleUSBHostDeviceUserClient AppleUSBHostFrameworkDeviceClient
HID	IOHIDEventServiceUserClient IOHIDEventServiceFastPathUserClient IOHIDResourceDeviceUserClient

Table 3: Special arguments recognized by KextFuzz

variable type	true positive	false positive
corefoundation variable	7	0
resource variable	26	3
pointer	19	9

Figure 8 shows the coverage of each configuration. The solid line is the average coverage of three rounds and the shadow shows the value range. We follow the fuzzing testing suggestions [30, 36] to calculate the statistical significance with the Mann Whitney U-test p-value. Overall, the result shows that KextFuzz gets the best performance when enabling all its components (i.e., the KF-En-K&U). Each component has its contribution to fuzzing.

The entitlement filter has a decisive impact on the range of the code that can be tested. There is a clear gap between the max coverage reached by KF-K and KF-En-K. This is because kexts patched by the entitlement filter treat KextFuzz as a privileged process and allow it to invoke sensitive functions. The Audio-related and HID-related group results show that code protected by entitlements can occupy a high proportion of the driver. This part of the code is hardly tested previously, leaving security risks for the system. The Graphic-group does not have coverage improvement when entitlement checks are removed because calling the privileged interfaces in this group not only needs the entitlements but also interface information.

The ability to extract interface knowledge from user space libraries makes KextFuzz can generate high-quality test cases and show greater potential when fuzzing complex interfaces. Compared with KF-En-K, KF-En-K&U reaches more code. This is because KF-En-K only knows the arguments' size while KF-En-K&U is aware of their types and semantics. An example of the interface specification generated by KextFuzz is in Appendix Listing 3. We manually verified the special variables recognized by KF-EN-K&U. The result is shown in Table 3. The false positive rate of the pointer type is relatively high because KextFuzz supposes the unknown external function return value are pointers. We also manually verified the result of KextFuzz interface analyzer with the open-sourced kexts. Among 89 arguments from 68 interfaces exported by nine open-sourced kext clients, KextFuzz infers 86 argument types correctly.

SyzGen also underperforms KF-EN-K&U. One of the reasons is that it determines the value range and types of some variables incorrectly, e.g., inferring an int32 variable as a const value. SyzGen infers interface information with dynamic symbolic execution. It takes a system snapshot before starting symbolic execution. The constant state in the snapshot imports extra constraints and makes it difficult to analyze all possible situations. As a result, SyzGen makes overstrict assumptions about the value ranges of some variables.

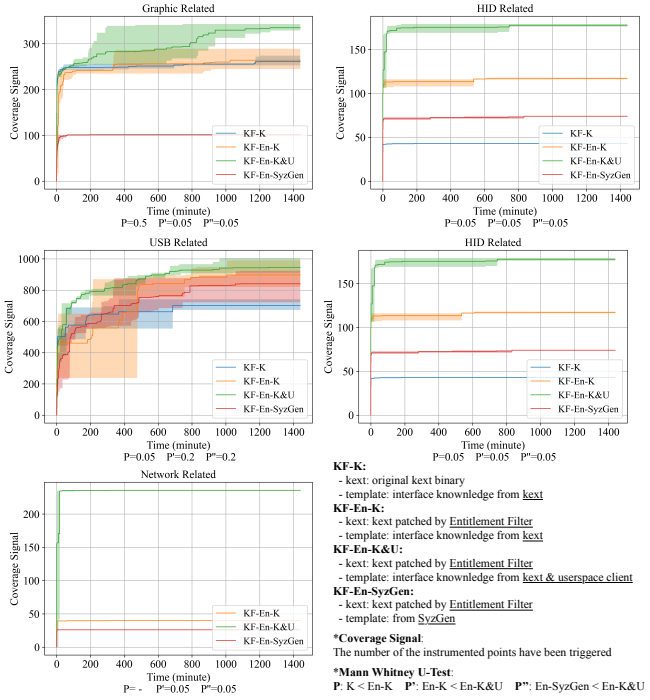

Figure 8: Coverage of KF-K, KF-En-K, KF-En-K&U, and KF-En-SyzGen

6.5 Bug Finding (RQ5)

The KextFuzz has found 48 unique macOS kernel bugs, including out-of-bounds and use-after-free vulnerabilities with serious impact. We follow the responsible disclosure process and report all the bugs to Apple. Five of the bugs have been fixed and assigned CVEs. Multiple of them have been committed to being fixed and assigned CVEs in upcoming security updates. One of them can prevent the device from rebooting without writing the firmware back to the device, which needs a physical connection. Table 5 in Appendix B shows the types and potential impacts of these bugs. For security considerations, some detailed information is hidden.

Each component of KextFuzz has contributed to bug finding. The experiment in Section 6.1 has already proved the Coverage Collector significantly improves the efficiency of bug detection. There are 18 bugs that can only be found with the Entitlement Filter. Two bugs can only be found and two bugs can be found faster with the Interface Analyzer.

Listing 2 shows a simplified example of a bug found by

KextFuzz. The kext client interface `createController` uses a CoreFoundation style dictionary input to create the kernel object `controller`. The vulnerable interface `setMask` can only work after the `controller` has been created. KextFuzz analyzes the client userspace wrappers and successfully found the input is a pointer pointing to a serialized CoreFoundation object. Therefore, it can create the `controller` during fuzzing. In another two cases, the vulnerable interface needs a structure-type argument that has a pointer-type member. KextFuzz recognizes the nested pointer type field successfully. Although the fuzzer can trigger the bug by using pointers as input heuristically, KextFuzz finds the bugs faster by implicitly declaring the pointer in the interface template.

```
client::createController(client* this, void* input){
  if (this->controller){ return ERROR; }
  v0 = OSUnserializeXML(input, ...);
  properties = TypeCast(v0, OSDictionary::metaClass);
  con = create_controller(properties);
  if (con){ this->controller = con; }
}
client::setMask(client* this, void* input){
  if (!this->controller){ /* vulnerable code */ }
}
```

Listing 2: An example of the bug found by KextFuzz

The bug finding result also demonstrates that fuzzing on Apple Silicon devices is necessary. Although macOS cross-compile some kexts into both x86_64 and arm versions, there are still Apple Silicon specific kexts, like GPU and BUS drivers. The cross-compiled kext binaries use the Fat Mach-O file structure, which packages the x86_64 and arm binary into one file. The others use the Mach-O file structure which has only a single version binary. We count the kexts distributed in macOS 11. There are 282 arm-only kext binaries, 115 x86_64-only binaries, and 313 binaries that have both architectures. KextFuzz finds thirteen bugs in arm-only kexts, which can not be found by fuzzing tools on x86_64 macOS. One of them is from an Apple SoC related driver. With this bug, attackers can corrupt the macOS secure boot firmware which stores in the SoC. In addition, iOS and iPadOS share more kexts with Apple Silicon macOS. The vulnerabilities have been assigned with CVEs also these two operating systems.

7 Generality Discussion

Although KextFuzz is developed for fuzzing macOS kexts, its method can be used in many other cases. The challenges faced by Apple Silicon kernel fuzzing are representative. Other security research, especially the studies targeting Commercial Off-The-Shelf (COTS) binaries, are facing a similar situation: closed-source code, strict runtime restrictions, and poor toolchain support. KextFuzz finds ways to solve them with minimized resource dependencies and deployment efforts.

The method of instrumentation based on instruction replacement works for binaries compiled with extra instructions. In fuzzing, the instructions used for mitigations (e.g. PAC, Canary, and Intel CET) can be replaced with instructions to collect coverage. Applicable targets include macOS kernel and userspace binaries, iOS applications, Android applications with PA, and most of the Linux binaries which generally adopt the Canary and CET. More generally, repeated instruction snippets can also be replaced. For example, function prologues and epilogues (e.g. `push rbp, mov rbp, rsp`) can be replaced to call an instrumented function and be executed later in the function, thus, avoiding using large trampolines.

We implemented a prototype to instrument Linux userspace binaries by replacing Canary and CET instructions. Since these instructions are usually used at the beginning or the end of the functions, we can do function-level instrumentation by replacing them. The instrumentation rate of our prototype is shown in Table 4. This is a POC experiment. The rate can be further increased with engineering efforts.

Table 4: Function instrumentation rates of replacing canary and cet instructions.

file	replacing canary	replacing cet
ls	21.15%	1.23%
bash	19.03%	8.61%
ssh	43.44%	5.05%
tar	32.72%	4.93%
watch	18.75%	7.69%
echo	20.00%	0.00%
diff	32.69%	1.94%
tnftp	35.79%	10.94%
dmesg	53.12%	9.38%
curl	37.86%	1.94%

The key idea of the Interface Analyzer in KextFuzz is to collect interface information from the userspace binaries rather than kext binaries. This idea also works for other closed-source targets with clients calling their interface, e.g., dynamic libraries and their executables, network services and their clients, hypervisors' virtual devices, and the guest OS drivers. The taint strategy of KextFuzz is also easy to be implemented or be migrated.

The entitlement check is a macOS / iOS customized mitigation, but we hope KextFuzz can inspire other works to dig into privilege checks (e.g., SELinux) to test deep code.

8 Related Work

8.1 Kernel Fuzzing

The kernel is an attractive target for both attackers and security researchers. Syzkaller [18] is one of the most famous kernel fuzzing tools. It generates system sequences based on system call templates. kAFL [41] utilizes Intel-PT to perform coverage-guided fuzzing. HFL [29] enables hybrid fuzzing on the Linux kernel. Digtool [38] detects Windows kernel bugs with a customized virtualization monitor. SyzGen [21] and IMF [28] can fuzz macOS with interface specifications

on intel macOS based on kext invocation traces. Our work further enables greybox and interface-aware fuzzing on Apple Silicon macOS, which is a more demanding and harsher environment for dynamic testing.

8.2 Coverage Collection

Code coverage feedback is an essential component of greybox fuzzing because it can guide the fuzzer to explore deeper paths. For open-sourced programs, source code level instrumentation is the most convenient choice. However, for closed-source softwares, it is not easy to get coverage feedback. Researchers have proposed some solutions, including utilizing hardware features, dynamic binary instrumentation, binary rewriting, and others. Intel introduced a hardware feature named Intel Processor Trace (Intel-PT). Many fuzzers [39–42] use it to collect coverage. However, the latest macOS on Apple Silicon and many systems running on customized hardware (e.g. embedded systems) do not provide similar features. Dynamic binary instrumentation can modify instructions in memory during execution. Representative work includes DynamoRIO [3], Pin [32] and Valgrind [34]. This method monitors the code execution and instrument executables at runtime, which makes them work soundly but suffer from high overhead.

Binary rewriting is also a useful method. However, existing tools are not stable enough or not practical when applied to the kernel. RetroWrite [25] recovers the relative offset after instrumentation with relocation information of position-independent code (PIC). However, it does not support C++ programs [16]. Existing works [27, 47] also point out it can not recover all the offsets correctly, which leads to crashes. STOCHFUZZ [47] uses an incremental rewriting method, which has to crash and re-instrument the program many times before determining the final strategy, which is time-consuming and needs significant manual effort when applying to the kernel. e9patch [26] uses a clever method to instrument the binaries with trampolines, but it only supports x86_64 binaries and needs a complex memory layout.

8.3 Interface-aware Fuzzing

Being aware of the system call interfaces is essential for kernel fuzzing. DIFUZE [23] performs static analysis on the source code of the Linux drivers to extract information about driver interfaces. KSG [44] improves the accuracy of driver interfaces analysis by taking the driver modules' dynamic registration information into consideration. Moonshine [37] and HFL [29] analyzed the relationship and dependencies between syscalls by analyzing the common variables read and written by them. These works provide abundant ideas of interface analysis but all of them rely on the source code.

For the Windows kernel, NTFuzz [22] collects type information from the Windows SDK header files and then performs static analysis on the SDK binaries to infer the type information of system calls. However, it does not support identifying driver interfaces that are nested behind system calls and are usually more complicated.

On macOS, IMF [28] and SyzGen [21] infer the dependencies of kext interfaces by analyzing the kext invocation logs. IMF hooks the userspace applications by dynamic library injection. SyzGen further uses dynamic symbolic execution to infer the argument type. LLDBFuzzer [11] does not infer the interface structure directly but tries to generate valid input by hooking fuzzing (or passive fuzzing), i.e., hook and mutate the kext invocation raised by real applications. These works use system runtime kext interactions to facilitate testcases generation. However, kext invocations raised by applications are limited, especially for the infrequently used kexts.

8.4 Apple Security

Except for kext fuzzing, there are other iOS and macOS security research works in recent years. p-joker [13] uses heuristic methods to extract useful information from kernel binaries to facilitate binary analysis. iDEA [20] summarizes the patterns of some kext vulnerabilities and can find them automatically based on their static method. In userspace, Inter Process Communication (IPC) is a large attack surface as well as a focus of research. Kun et al. [46] combined static and dynamic analysis to construct high-quality IPC messages for fuzzing IPC services without source code. Kobold [24] and iService [45] can identify confused deputy problems in AppleOS IPC.

9 Conclusion

macOS is an attractive target for adversaries. However, it is more challenging to discover vulnerabilities in macOS (especially for the one running on Apple Silicon) than in other systems, e.g., Linux or Windows, due to its closed-source nature and customized hardware environment, as well as widely deployed mitigations. In this paper, we showed that the mitigations deployed by macOS can be leveraged to (1) rewrite kernel binaries to track code coverage or disable mitigations, (2) test kexts that are accessed infrequently, and (3) infer the input format. Based on this insight, we propose a solution KextFuzz to automatically discover vulnerabilities in macOS kernel extensions. Our prototype has found dozens of vulnerabilities in the latest macOS, showing this solution is effective.

Acknowledgements

We would like to sincerely thank all the anonymous reviewers and our shepherd for their valuable feedback that greatly helped us to improve this paper. This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), and Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

References

[1] Apple document. entitlements. https://developer.apple.com/documentation/bundleresources/entitlements. Accessed: 2022-05-11.

[2] Cve-2022-22675. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22675. Accessed: 2022-06-02.

[3] Dynamorio: Dynamic instrumentation tool platform. https://github.com/DynamoRIO/dynamorio. Accessed: 2022-05-22.

[4] Dyninstapi: Tools for binary instrumentation, analysis, and modification. https://github.com/dyninst/dyninst. Accessed: 2022-05-22.

[5] High-performance binary-only instrumentation for afl-fuzz. https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu. Accessed: 2022-05-12.

[6] Intel® 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 2022-11-06.

[7] The i/o registry | apple documentation archive. https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/TheRegistry/TheRegistry.html. Accessed: 2022-06-07.

[8] kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html. Accessed: 2022-05-14.

[9] The kernel address sanitizer (kasan). https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html. Accessed: 2022-05-14.

[10] Kernel debug kit 12.5 build 21g5027d. https://developer.apple.com/download/all/. Accessed: 2022-05-20.

[11] Lldbfuzzer: Debugging and fuzzing the apple kernel. https://www.trendmicro.com/en_us/research/19/h/lldbfuzzer-debugging-and-fuzzing-the-apple-kernel-with-lldb-script.html. Accessed: 2022-05-12.

[12] Operating system integrity | apple developer documentation. https://support.apple.com/zh-cn/guide/security/sec8b776536b/web. Accessed: 2022-06-06.

[13] p-joker: ios/macos kernelcache/kexts analysis tool. https://github.com/lilang-wu/p-joker. Accessed: 2022-05-20.

[14] Preparing your app to work with pointer authentication | apple developer documentation. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication. Accessed: 2022-06-06.

[15] Process injection: Breaking all macos security layers with a single vulnerability. https://sector7.computest.nl/post/2022-08-process-injection-breaking-all-macos-security-layers-with-a-single-vulnerability/. Accessed: 2022-12-06.

[16] Retrowrite. https://github.com/HexHive/retrowrite. Accessed: 2022-11-06.

[17] Strong arming with macos: Adventures in cross-platform emulation. https://blogs.blackberry.com/en/2021/05/strong-arming-with-macos-adventures-in-cross-platform-emulation. Accessed: 2022-06-06.

[18] syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller. Accessed: 2022-05-14.

[19] Triton: a dynamic binary analysis library. https://github.com/JonathanSalwan/Triton. Accessed: 2022-11-06.

[20] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. idea: Static analysis on the security of apple kernel drivers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1185–1202, 2020.

[21] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 749–763, 2021.

[22] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693. IEEE, 2021.

[23] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[24] Luke Deshotels, Costin Carabas, Jordan Beichler, Răzvan Deaconescu, and William Enck. Kobold: Evaluating decentralized access control for remote nsxpc methods on ios. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1056–1070. IEEE, 2020.

[25] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[26] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[27] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092, 2020.

[28] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[29] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[31] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[33] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700, 2021.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[35] Pádraig O'sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in cots software with binary rewriting. In *IFIP International Information Security Conference*, pages 154–172. Springer, 2011.

[36] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. My fuzzer beats them all! developing a framework for fair evaluation and comparison of fuzzers. In *European Symposium on Research in Computer Security*, pages 173–193. Springer, 2021.

[37] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[38] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A Virtualization-Based framework for detecting kernel vulnerabilities. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 149–165, 2017.

[39] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[40] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. 2020.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL:hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[42] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network fuzzing with incremental snapshots, 2021.

[43] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[44] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. KSG: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, 2022.

[45] Yizhuo Wang, Yikun Hu, Xuangan Xiao, and Dawu Gu. iservice: Detecting and evaluating the impact of confused deputy problem in appleos. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 964–977, 2022.

[46] Kun Yang, Hanqing Zhao, Chao Zhang, Jianwei Zhuge, and Haixin Duan. Fuzzing ipc with knowledge inference. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 11–1109. IEEE, 2019.

[47] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.

A Interface Specification Case Study

```
1  //code in userspace binary
2  v11 = CFArrayCreate(0, (const void **)&cfstr, ...);
3  if ( v11 ) {
4    v13 = IOCFSerialize(v12, 1);
5    if ( v13 ) {
6      len = CFDataGetLength(v13);
7      input = malloc(Len + 8);
8      // a resource variable
9      *((int *)input) = *(this + 112);
10     BytePtr = CFDataGetBytePtr(v13);
11     // a cfarray variable
12     memcpy(input + 2, BytePtr, v15);
13     IOConnectCallMethod(v6, 9, input, len+8, ...);
14   }
15 }
16
17 //the interface specification generated by KextFuzz
18 resource key_global0[int32]
19
20 IOConnectCallMethod$sel9_v0(..., sel const[9], ...,
21                   input ptr[in, sel9_v0_istr], ...)
22 sel9_v0_istr {
23     key_global0_0   key_global0
24     align_1 const[0, int32]
25     cfvar_2 sel9_v0_istr2
26 } [packed]
27 sel9_v0_istr2  {
28     prefix stringnoz["<array>"]
29     entry array[cfstring]
30     suffix string["</array>"]
31 } [packed]
32 cfstring {
33     prefix stringnoz["<string>"]
34     value stringnoz[IOSurfaceRootUserClient_strs]
35     suffix stringnoz["</string>"]
36 } [packed]
37
38 IOSurfaceRootUserClient_strs = "aSDb","aS0d","dSpd",
39 "dSld","P^UdS", "dS\d","aSTk","fS<k","NhS\l","fSD1",
40 "aS(d","aS\0","gSpf","gSXg","IOSurfaceHeight",
41 "IOSurfaceBytesPerRow","IOSurfaceBytesPerElement",
42 "IOSurfaceElementWidth","IOSurfaceElementHeight",
43 "IOSurfaceOffset","IOSurfacePlaneInfo",
44 "IOSurfacePlaneHeight","IOSurfacePlaneBytesPerRow",
45 "IOSurfacePlaneOffset","IOSurfacePlaneSize",
46 ...
```

Listing 3: An example of the interface specification generated by KextFuzz

Line 17 to line 46 in Listing 3 shows an example of the kext interface specification generated by KextFuzz. KextFuzz performs taint analysis on the userspace binary code snippet from line 1 to line 15 and finds the input argument for this interface is a struct with three fields: a `int32` type resource variable, an align field and a CoreFoundtion style array expression. For the string elements in the array, KextFuzz uses the strings extracted from the userspace binary as their seed (line 38 - line 46).

B Bug Finding Result

Table 5: KextFuzz finds 48 unique kernel bugs macOS kexts. Some detailed information is hidden due to security considerations.

Panic Reason	Kext	Impact	Status
Firmware Destruction	***	Exploitable	To be fixed
Firmware Destruction	***	Exploitable	To be fixed
Memcpy Check Failed	***	DoS	CVE-2022-32829
IOMemory Error	IOUserEthernet	DoS	Reported
IOMemory Error	AppleMobileFileIntegrity	DoS	Confirmed
IOMemory Error	IOUSBHostFamily	DoS	Reported
IOMemory Error	IOUSBHostFamily	DoS	Reported
Null Pointer	IOUSBHostFamily	DoS	Reported
DMA Error	IOUSBHostFamily	DoS	Confirmed
DMA Error	IOUSBHostFamily	DoS	Confirmed
Null Pointer	IOUSBHostFamily	DoS	Reported
IOMemory Error	AppleSystemPolicy	DoS	Reported
IOMemory Error	AppleSystemPolicy	DoS	Reported
IOMemory Error	AppleSystemPolicy	DoS	Reported
IOMemory Error	AppleSystemPolicy	DoS	Reported
Assert Failure (misc)	IOSlaveProcessor	DoS	Reported
Assert Failure (misc)	IOMobileGraphicsFamily	DoS	Reported
Assert Failure (misc)	AppleSPU	DoS	Confirmed
Overflow Detected	***	Likely Exploitable	Reported
Null Pointer	IOHIDFamily	DoS	Reported
IOMemory Error	IOHIDFamily	DoS	Reported
IOMemory Error	IOUSBDeviceFamily	DoS	Reported
IOMemory Error	IOUSBDeviceFamily	DoS	Reported
UAF	***	Exploitable	CVE-2022-42806
Null Pointer	IOAVBFamily	DoS	Reported
LLC Bus Error	AppleH13CameraInterface	DoS	Confirmed
OOB Read	***	Exploitable	CVE-2022-32949
UAF	***	Exploitable	CVE-2023-28181
DMA Error	AppleUSBUserHCI	DoS	Reported
Null Pointer	IOUSBHostFamily	DoS	Confirmed
Refcount Error?	***	Likely Exploitable	Reported
DMA Error	IONVMeFamily	DoS	Reported
Null Pointer	IOgPTPPlugin	DoS	Reported
LLC Bus Error	IOMobileGraphicsFamily	DoS	Reported
Assert Failure (misc)	AppleSEPManager	DoS	Reported
Assert Failure (misc)	AppleSEPManager	DoS	Reported
Assert Failure (misc)	AppleSEPManager	DoS	Reported
Data Corruption	***	Exploitable	Reported
Data Corruption	***	Exploitable	Reported
Null Pointer	RTBuddy	DoS	Reported
Null Pointer	AppleSPU	DoS	Reported
Assert Failure (misc)	ApplePMP	DoS	Reported
IOMemory Error	AppleSSE	DoS	Reported
Null Pointer	IOHIDFamily	DoS	Confirmed
IOMemory Error	IOHIDFamily	DoS	Reported
Null Pointer	IOUserEthernet	DoS	Reported
OOB Read	***	Exploitable	Reported
OOB Write	***	Exploitable	CVE-2023-27936