iDEA: Static Analysis on the Security of Apple Kernel Drivers

Xiaolong Bai
bxl1989@gmail.com
Orion Security Lab, Alibaba Group

Luyi Xing*
luyixing@indiana.edu
Indiana University Bloomington

Min Zheng
zhengmin1989@gmail.com
Orion Security Lab, Alibaba Group

Fuping Qu
fuping.qfp@alibaba-inc.com
Orion Security Lab, Alibaba Group

ABSTRACT

Drivers on Apple OSes (e.g., iOS, tvOS, iPadOS, macOS, etc.) run in the kernel space and driver vulnerabilities can incur serious security consequences. A recent report from Google Project Zero shows that driver vulnerabilities on Apple OSes have been actively exploited in the wild. Also, we observed that driver vulnerabilities have accounted for one-third of kernel bugs in recent iOS versions based on Apple's security updates. Despite the serious security implications, systematic static analysis on Apple drivers for finding security vulnerabilities has never been done before, not to mention any large-scale study of Apple drivers.

In this paper, we developed the first automatic, static analysis tool *iDEA* for finding bugs in Apple driver binaries, which is applicable to major Apple OSes (iOS, macOS, tvOS, iPadOS). We summarized and tackled a set of Apple-unique challenges: for example, we show that prior C++ binary analysis techniques are ineffective (i.e., failing to recover C++ classes and resolve indirect calls) on Apple platform due to Apple's unique programming model. To solve the challenges, we found a reliable information source from Apple's driver programming and management model to recover classes, and identified the unique paradigms through which Apple drivers interact with user-space programs. *iDEA* supports customized, pluggable security policy checkers for its security analysis. Enabled by *iDEA*, we performed the first large-scale study of 3,400 Apple driver binaries across major Apple OSes and 15 OS versions with respect to two common types of security risks – race condition and out-of-bound read/write, and discovered 35 zero-day bugs. We developed PoC and end-to-end attacks to demonstrate the practical impacts of our findings. A portion of the bugs have been patched by recent Apple security updates or are scheduled to be fixed; others are going through Apple's internal investigation procedure. Our evaluation showed that *iDEA* incurs a low false-positive rate and time overhead.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

*Corresponding author.

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3423357

KEYWORDS

Apple; Kernel Drivers; iOS; iPadOS; tvOS; macOS; Static Analysis; Vulnerability Detection

ACM Reference Format:
Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. 2020. *iDEA*: Static Analysis on the Security of Apple Kernel Drivers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3372297.3423357

1 INTRODUCTION

The Apple OSes, i.e., tvOS, iOS, iPadOS, watchOS, macOS, feature a hybrid kernel called *XNU* (an abbreviation of "X is Not Unix"), which includes network stacks, IPC mechanisms, device drivers, etc. Among them, device drivers act as a bridge between software and hardware (e.g, the camera, microphone, BLE devices, USB, speaker, sensors, etc.).

Security risks with Apple drivers. Since drivers on Apple OSes (called *Apple drivers* in this paper) generally run in the kernel space, vulnerabilities found in drivers can have serious security impacts: exploiting driver vulnerabilities on Apple OSes enables a malicious user-space application to execute arbitrary code or read/write arbitrary memory address in the kernel space, or even completely control Apple OS kernel [43]. By examining Apple's security updates [8] from iOS 8 through the recent iOS 13.4.1, we found there are 74 CVEs related to Apple drivers, which account for approximately one-third of all 231 reported Apple kernel vulnerabilities. In the wild, unfortunately, recent evidence shows that Apple driver vulnerabilities have been actively exploited to attack real-world users. In August 2019, a report from Google Project Zero [44] found that, for a few years, malicious websites have been silently installing monitoring implants on the iPhones of website visitors. Among all 14 vulnerabilities that have been exploited, four are driver vulnerabilities that give the malicious websites kernel privilege on the victims' iPhones. With the kernel privilege, the adversary can take full control of the victim device, such as to install malicious apps, steal users' private data like messages and photos, inspect applications' internal data, etc. Further, Apple driver vulnerabilities have also seen serious usage in APT attacks: for example, an iOS malware called Pegasus was found to be exercised against high-value victims as far back as 2016 [47].

Although the community has seen automatic tools [37, 60] for Apple kernel analysis, these tools cannot be easily applied to Apple drivers. Prior works from industry performed analysis on iOS or macOS drivers [27, 28, 85], which, however, largely relied on manual reverse engineering and OS-specific heuristics. For example, Ryuk [85] relied on debug symbols only available in macOS drivers, and its security analysis is mainly manual. In the absence of a

general, automatic analysis technique, these approaches are rather time-consuming, even error-prone and do not scale since recent Apple OSes all come with a plethora of drivers (e.g., more than 170 drivers on iOS, 180 on iPadOS, 350 on macOS, see Section 5). Also, as Apple devices (iPhone, iPad, Apple TV, Mac, etc.) increasingly adopt new or updated peripheral devices (new-generation cameras, speakers, microphones, graphic devices, sensors, etc.), device drivers are always updating. Despite the serious security impact, we are unaware of any systematic, automatic security analysis techniques that are general to all major Apple OSes (iOS, tvOS, macOS, iPadOS, etc.), not to mention any large-scale study on Apple drivers.

New challenges. Apple drivers are programmed in C++ but usually do not come with source code, debug symbols (symbols only available for a portion of drivers on macOS), or RTTI information [3]. Hence, Apple driver analysis has to tackle stripped binaries (no debug symbols) whose C++ abstractions (e.g., classes, member functions, inheritance hierarchies) are lost in compilation, the recovery of which is essential for an automatic analysis [49, 59, 67]. However, prior static analysis for C++ binaries [31–33, 49, 59, 67] are ineffective in analyzing Apple drivers due to Apple's unique driver programming model (detailed below). Also, prior techniques for driver analysis on non-Apple platforms cannot work for Apple due to the Apple-unique driver management (e.g., driver interactions with user space), programming model, and even availability of source code (see Section 5.2). For example, driver analysis on Linux, such as Dr. Checker [51], can leverage their C-language source code, while Apple drivers are closed-source and programmed in C++. In the following, we summarize the major challenges in the static analysis of Apple drivers.

• *Recovering C++ classes.* Apple driver features a unique programming model: Apple kernel and drivers can instantiate objects of driver C++ classes simply using the class name (by calling kernel API `OSMetaClass::allocClassWithName()`, see Section 2.2). This is a dynamic allocation feature, essential for Apple driver management at runtime (Section 2.2), but introducing a problem to driver analysis: state-of-the-art techniques for recovering C++ classes [49, 59, 67] (recovering vtables, inheritance hierarchies, etc.), essential for automatic C++ binary analysis (e.g., resolving indirect calls and building control-flow graphs), are made ineffective.

There are two reasons for the issue. First, constructors of C++ classes are often removed from Apple driver binaries since class instantiation can leverage the above general kernel API using a class name, without calling constructors of specific classes. This, however, renders prior constructor-analysis-based approaches [59, 67] ineffective: they identify constructors in the binary, which further help identify inheritance hierarchies, vtables, etc. Second, recent approaches rely on the identification of vtable assignment through specific code patterns [31–33, 49, 59], which helps recover class hierarchies and identify assigned vtables for constructed objects (so to resolve indirect calls on the objects). However, these approaches are ineffective for Apple since Apple kernel leverages a runtime map (with vtables of all driver classes, see `MetaClass` map in Section 2.2) to select the right vtable and assign it to an object. Without such runtime information and analysis of Apple's dynamic allocation mechanism, prior approaches cannot work on Apple platforms.

Due to the above reasons, in our evaluation of the state-of-the-art work *Marx* [59] (designed to recover C++ classes and resolve virtual-function calls) on a set of 362 Apple driver binaries with 8,217 classes, *Marx* incurs a high false-positive rate in recovering classes (34% false positives compared to the 100% precision of our tool to be introduced in this paper); *Marx* resolves only 20% of indirect calls compared to the 66% resolved by our tool (see Section 5.3).

• *Finding driver entry points.* Entry points are the interface functions that an Apple driver exposes to the user-space programs (see Section 2.3) which are essential starting points of driver analysis and bug detection [51]. On Windows and Linux, how to find entry points is well known: they are located in specific data structures, e.g., `WDF_DRIVER_CONFIG` [54] and `file_operations` [51]; these data structures can be located in certain functions used to register drivers to the kernel (i.e., `WdfDriverCreate` [55] and `register_chrdev` [51]). However, there is no known, public approach that enables systematic discovery of entry points for Apple drivers since Apple has unique, proprietary driver interface management to govern how drivers interact with user-space programs (see Section 2.3).

Our work. In this paper, we present *iDEA* (an alias for *Apple Driver Security Analyzer*), an automatic, static security analysis tool for Apple driver (binaries), which is applicable to major Apple OSes – iOS, tvOS, macOS, and iPadOS. *iDEA* tackles the new challenges (summarized above) and is capable of inter-procedural analysis on Apple drivers, i.e., control-flow and data flow analysis starting from Apple-unique driver entry points. *iDEA* is implemented as a modular framework, where customized security policies can be specified and checked for identifying driver vulnerabilities.

To solve the challenges in Apple driver analysis, we found that one should find a reliable information source from Apple's driver programming and management model to recover classes, and identify the unique paradigms how Apple drivers interact with user-space programs. In particular, we demystified Apple's unique driver programming model, driver registration and interface management (see Section 2), which we first systematized through analyzing Apple's driver development framework I/O Kit [5] and studying real-world Apple drivers. Specifically, we show that each driver registers information of its class (name, inheritance hierarchy, vtable, size, etc.) to the kernel; such information is essential since it enables Apple kernel to conveniently instantiate the driver's class through its class name at runtime (see Section 2). Such an essential registration procedure provides us a reliable source to recover driver classes: just like how the kernel gathers the class information, our analysis follows the same procedure to recover classes (see Section 3.1). Further, we identified two general ways for Apple drivers to provide interfaces to user-space programs and how Apple organizes the interfaces. We show that finding these interfaces from driver binaries is the starting point of static driver analysis, enabling us to build an inter-procedural control flow (Section 3.2).

We applied *iDEA* to perform a large-scale security analysis on 3,400 Apple driver binaries across 15 OS versions, ranging from the earlier iOS 8, macOS 10.13, iPadOS 13.1, and tvOS 13.2, to their latest versions (Section 5). With two security policy checkers we implemented on *iDEA* for identifying common driver security risks, including race condition and out-of-bound read/write (Section 4), we successfully detected more than 40 previously unknown driver

vulnerabilities. This was done with high precision (92%) and low time overhead (e.g., analyzing all 362 drivers on macOS 10.15.6 within 14 hours).

To demonstrate the practical impacts of the vulnerabilities we found, we implemented PoC exploits: e.g., exploiting one vulnerability, an unpriviledged user-space program on macOS 10.15.4 (the latest at the time of the finding) successfully gains root privilege and runs arbitrary code in the kernel space (see video demo [2]). We reported all vulnerabilities to Apple, who acknowledged our findings, and issued CVEs. With our reports, Apple has fixed a few vulnerabilities in early 2020, and more fixes are scheduled.

Contributions. We summarize the contributions as follows:

• *iDEA* is the first automatic, static bug finding tool for Apple driver binaries that applies to major Apple OSes. We identified new, unique challenges in Apple driver analysis and developed novel, automatic, systematic methods to address them.

• We show that *iDEA* supports pluggable security policy checkers; we developed two checkers on *iDEA* which are capable of large-scale security analysis on Apple drivers.

• We performed the first large-scale security analysis on Apple driver binaries, which led to the discovery of 35 zero-day vulnerabilities. We implemented a few proof-of-concept exploits, which demonstrated serious security implications. We plan to release the source code of *iDEA* (15,000 lines of source code in Python) [2].

• Our evaluation shows *iDEA* incurs a low false positive rate and time overhead.

2 BACKGROUND

This section introduces (1) the mechanism Apple kernel uses to facilitate driver registration, which is essential for its runtime driver management, and (2) how Apple drivers interact with user-space programs.

2.1 Driver Programming Model.

Apple has a unique model for regulating driver programming and how driver functions are exposed to the kernel and user space. In the model, each driver is implemented as a C++ class – also called *driver class* in this paper; correspondingly, we simply call the driver class' instance *driver instance*. The driver class must inherit a kernel class `IOService` (defined in Apple's I/O Kit [5], a kernel framework for Apple driver programming), and accordingly implement a set of virtual functions inherited (Table 1). [1] These functions are called by the kernel and referred to as *driver-callbacks*: e.g., the kernel calls `start()` on a *driver instance* to start the driver after it is instantiated by the kernel. In addition, Apple does not allow driver classes to define customized constructors/destructors; instead, drivers adopt pre-defined marcos `OSDefineDefaultStructors` [4] (defined in I/O Kit) to generate constructors/destructors. Note that the constructor will be removed from driver binary if there is no explicit invocation in the driver's code space.

A driver binary (in Mach-O format [83]) can bundle multiple drivers with their driver classes and utility implementation. Also, a driver binary is packaged with a configuration file (in plist format [6]), which specifies each driver's name, class name, and the types of hardware devices it handles (e.g., ethernet adapter, human interface device, USB storage device, etc.). Such information enables the kernel to know which driver to use when handling a certain device and what its class is when instantiating a *driver instance*. Installed driver binaries and their configuration files can be found under `/System/Library/Extensions` on Apple OSes.

[1] C++ on Apple platforms mandates single inheritance.

Table 1: Sample *driver-callbacks*

Callback Name	Description
`start()`	the driver is about to start
`stop()`	the driver is about to stop
`initWithTask()`	the driver is about to be initialized
`setProperties()`	set the properties of the driver
`newUserClient()`	create the driver's UserClient instance

2.2 Registering Drivers to Apple Kernel

All drivers with their driver class information need to be registered to the kernel. To access a hardware device, e.g., a plugged USB key, or the built-in microphone, the kernel first figures out which driver to use (based on hardware type), then leverages a *driver instance* of the driver to operate the device (by calling methods on the instance). The kernel instantiates *driver instances* when a driver binary is loaded to the kernel, and maintains all instances in a runtime pool, ready to use (see Figure 2).

To facilitate the management and easy instantiation of *driver instances*, especially when a driver is on demand (e.g., a new USB key is inserted, and, thus, a new `IOUSBDevice` *driver instance* is needed to access the key), Apple kernel maintains driver class information (class name, size, inheritance relationship, etc.). Based on such information, the kernel can easily instantiate a *driver instance* through a class name (by calling `OSMetaClass::allocClassWithName()` kernel API, see details below). This is extensively used in Apple kernel and driver programming.

Registering class information through InitFunc. To facilitate the registration of driver class information to the kernel, the Apple driver compiler places a special section called "__mod_init_funcs" in a driver binary. This section contains a list of function pointers to driver initialization functions, called *InitFuncs*, with each *InitFunc* corresponding to a particular driver class in the binary. *InitFuncs* are executed automatically by the kernel when the binary is loaded at runtime, to register driver class information to the kernel (see an example below). Specifically, each *InitFunc* wraps the information of a driver class into an object of a `MetaClass` – a subclass that inherits from the kernel class `OSMetaClass`. Each InitFunc/driver class corresponds to a particular `MetaClass` (e.g., `IOSurfaceRoot::MetaClass` for the driver class `IOSurfaceRoot`, see the example below).

At runtime, the kernel maintains a map of `MetaClass` objects for all drivers registered to the kernel. These objects enable Apple kernel to easily instantiate driver (class) instances through the aforementioned API `allocClassWithName()` (using a class name). Specifically, to create an instance of a certain driver, the API internally finds the corresponding `MetaClass` object from the map, and invokes its `MetaClass::alloc()` (a virtual function implemented in each `MetaClass`), which creates the *driver instance*. To create a driver instance, `MetaClass::alloc()` internally allocates memory for the instance based on the size of the driver class, and assigns a vtable to the instance.

An example of InitFunc. Figure 1 illustrates an *InitFunc*, which creates a `MetaClass` object. Specifically, register X0 (Line 1-2) holds

Table 2: Names and descriptions of *user-entries*

Name	Description	Corresponding system APIs
`externalMethod()`	provide methods to user-space programs	IOConnectCallMethod
`getTargetAndMethodForIndex()`	provide methods to user-space programs (legacy user-entry)	IOConnectCallMethod
`getAsyncTargetAndMethodForIndex()`	provide methods that return results asynchronously (legacy user-entry)	IOConnectCallAsyncMethod
`getTargetAndTrapForIndex()`	similar to `getTargetAndMethodForIndex` (legacy user-entry)	IOConnectTrapX
`clientMemoryForType()`	share memory with user-space programs	IOConnectMapMemory
`registerNotificationPort()`	allow user-space programs to register for notifications	IOConnectSetNotificationPort
`setProperty()`	set runtime property of the userclient	IOConnectSetCFProperty
`clientClose()`	stop using the userclient	IOServiceClose

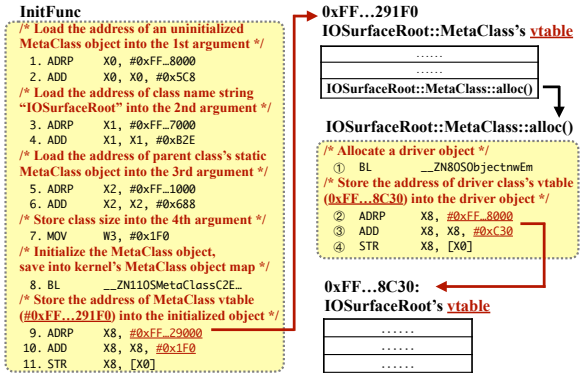

Figure 1: An example of InitFunc and its MetaClass

a pointer to the uninitialized `MetaClass` object (a static object in the binary's data section); register X1 (Line 3-4) holds pointer to the string of driver class name; register X2 (Line 5-6) holds pointer to the static `MetaClass` object of the driver class' parent class; register W3 holds the size of the driver class; taking X1, X2 and W3 as arguments, at Line 8, *InitFunc* calls constructor `OSMetaClass::OSMetaClass()` (through `BL` instruction) to instantiate the `MetaClass` object pointed to by X0. After instantiation, at Line 9-11, the `MetaClass`'s vtable is assigned to the (starting address of the) `MetaClass` object. In Section 3, we will elaborate on our analysis on *InitFunc* to recover driver class information, including its vtable used by `MetaClass::alloc()` for instantiating the driver class.

2.3 Interacting with User-space Programs

A driver with its driver class and functions runs in the kernel space and is accessible by the kernel, but it is not directly exposed to user space programs. To serve user-space requests, a driver class typically has a companion class, called *UserClient*, that acts like its delegate and is exposed to the user space (through system API calls, see below). A driver's *UserClient* (class) inherits from a generic class `IOUserClient` (also defined in I/O Kit) and implements a set of virtual functions inherited, as listed in Table 2. We call these functions *user-entries* since although they run in the kernel space they are exposed to user-space programs (through a set of system APIs) for accessing driver functionalities.

Figure 2 illustrates the typical process when a user-space program interacts with a driver through its *UserClient*. First, the (user-space) program obtains a handle to an instance of the driver in need (`IOUSBDevice`). This is through a system API call (`IOServiceGetMatchingService`) with the driver class name "IOUSBDevice" specified; in response, the kernel finds the driver instance from its runtime information pool that maintains instances of all drivers registered and returns its handle (see Section 2.2).

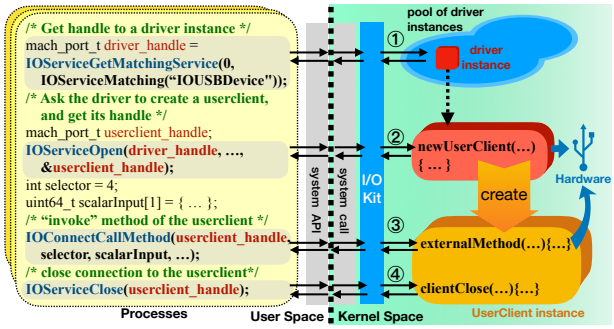

Figure 2: The interactions between user-space programs and drivers

Then (step ②), using the handle, the user-space program obtains another handle to the *UserClient* instance of the driver. This is through a system API call (`IOServiceOpen`); in response, the kernel calls `newUserClient` on the driver instance, which instantiates a *UserClient* object (of class `IOUSBDeviceUserClient`).

To access the driver using the *UserClient* handle, the user-space program can trigger a *user-entry* through a particular system API (see Table 2). Then, a *user-entry* can further invoke functions of the driver instance or implement driver functionalities itself. An example is shown in Figure 2 that triggers a *user-entry* `externalMethod` (step ③). Specifically, after system API `IOConnectCallMethod` is called from the user-space program, the kernel invokes `externalMethod` on the *UserClient* instance. Note that `externalMethod` uses the *selector*, an argument specified in the system API call, to select (execute) specific functionality the user space requests (e.g., GetDeviceInformation). To fulfill the request, `externalMethod` can opt for a few strategies: (1) invoke functions of the driver instance; (2) fulfill the functionality itself; and (3) return a function pointer of a *UserClient*-internal function to the kernel – the kernel will then invoke it. Also note that legacy *UserClient* may have `get*Target*ForIndex` *user-entries* (Table 2), which operate similarly to `externalMethod`.

InitFunc for UserClient. Like driver instances, a `UserClient` instance also needs to be created at runtime on demand (see Figure 2). Hence, just like a driver that has a corresponding *InitFunc* for the kernel to manage the class information and easily instantiate its instance (through a name), each `UserClient` has its *InitFunc* as well. Besides, driver developers may define other classes that inherit from kernel classes (e.g., `OSData`, `OSDictionary`, `OSSet`, etc.) defined in I/O Kit to handle data structures. These classes also come with corresponding *InitFuncs* in driver binaries. Section 3.1 will show our analysis on *InitFuncs* to recover classes from driver binaries. Figure 10 in Appendix outlines the inheritance relationship between driver related classes, including driver classes, `UserClients`, and kernel classes defined in I/O Kit. Note that all the classes have the same ancestor (`OSMetaClassBase` [19]).

3 ANALYSIS DESIGN

Overview. *iDEA* is an automatic security analysis framework that detects security bugs in Apple drivers using pluggable security policy checkers. To this end, *iDEA* first builds an inter-procedural control-flow graph (ICFG) starting from each entry point of a driver. *iDEA* then employs a set of pluggable security policy checkers which identify the policy violation along the control flow and raise warnings.

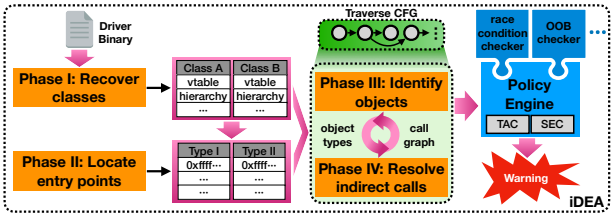

Figure 3: The analysis phases of *iDEA*

The general architecture of *iDEA* is outlined in Figure 3. *iDEA* first builds the ICFG in four phases. **Phase I** recovers driver class information (vtable, inheritance hierarchy, etc.) that is lost in compilation, a step needed for resolving indirect calls in C++ binaries [31–33, 49, 59, 67] (Section 3.1). This is done in our study by leveraging the mechanism by which Apple drivers are registered to the kernel, which is demystified in our study (Section 2.2). **Phase II** then locates all entry points of a driver in the binary since they are essential starting points of driver analysis (Section 3.2). For the first time, our study summarizes two types of entry points for Apple driver analysis and shows how they can be found. Starting from each entry point, *iDEA* traverses the driver to generate the ICFG, which tackles the challenges for identifying (the types of) driver class objects (**Phase III**) and resolving their virtual function calls (**Phase IV**). This incurs new challenges since the types and vtables of many objects cannot be identified using known approaches such as constructor-based analysis; this is because the objects are often instantiated simply through a class name (see Section 2.2) or instantiated elsewhere in the kernel invisible to the driver, etc. (see five scenarios in Section 3.3).

The ICFG(s) is produced once for a driver, which can then be leveraged by multiple policy checkers. Along the ICFG, *iDEA* invokes detection functions of a checker before and after each instruction. *iDEA*, as a framework also provides a few analysis clients that can be leveraged by checkers, i.e., taint analysis client (TAC) and symbolic execution client (SEC); this is done by integrating an off-the-shelf tool Triton [62, 65]. The details of our analysis and security policy checkers are introduced in the following sections. The implementation details are elaborated in Section 9 in Appendix.

3.1 Phase I: Recovering Driver Classes

Overview. Phase I aims to recover all classes from driver binaries, including vtables (the addresses in binaries) and inheritance hierarchies. The output of Phase I is (1) a `class-vtable` map, and (2) the inheritance hierarchies (rooted from generic kernel classes `IOService` and `IOUserClient`, see Section 2).

To recover classes, *our insight is to leverage the mechanism how driver classes are registered to Apple kernel*, an essential procedure in Apple driver management (see *InitFuncs* in Section 2.2). Specifically, just like how the kernel gathers the class information (names, sizes, vtables, hierarchies, etc.) from a binary, our analysis follows the same procedure to recover classes, i.e., by analyzing the *InitFunc* corresponding to each class in the binary. *This is a reliable information source to recover classes. In contrast, state-of-the-art works that recover C++ class mainly leverage heuristics or human-summarized code patterns [31–33, 49, 59, 67], which we show to be ineffective on Apple platforms (see our evaluation in Section 5.3).*

Analyzing InitFunc. As mentioned earlier (Section 2.2), an *InitFunc* corresponding to a certain driver class (or a `UserClient` class), wraps the class information into a `MetaClass` object. We take Figure 1 as a running example to show how *iDEA* recovers a driver class `IOSurfaceRoot` by analyzing its *InitFunc*.

Our analysis first looks for the driver class name and size from the registers that hold their pointers/values. From *InitFunc* at Line 8 we find the call to the constructor of the `MetaClass` (i.e., `IOSurfaceRoot::MetaClass`, corresponding to the driver class `IOSurfaceRoot`, see Section 2.2), and those registers used as its arguments indicate the (static) addresses of driver class name (`X1`), the `MetaClass` object to construct (`X0`), the `MetaClass` object of the driver class' parent class (`X2`), and class size (`W3`). To find the values in these registers, we first use backward slicing to find instructions that affect values in these registers; we then use forward constant propagation to identify the values in these registers.

Further, we performed forward analysis to find vtable for the driver class. The driver class' vtable is used in virtual function `IOSurfaceRoot::MetaClass::alloc()` of the `MetaClass` object – for assigning the vtable to the driver's instance (the `MetaClass`' alloc() is used to instantiate the driver's instance, see Section 2.2). Hence, to find the vtable, *iDEA* locates the `MetaClass` object's vtable when it is assigned to the `MetaClass` object (Line 9-11), and then finds the pointer to its `alloc()` function in the vtable (see Figure 1). What comes next is similar to a conventional constructor analysis: in `IOSurfaceRoot::MetaClass::alloc()`, we can find the vtable (Line ②-③) since it is stored into the starting address of the driver class instance to instantiate (④).

To recover driver class hierarchies, we can leverage the (pointer to) `MetaClass` object of the driver class' parent class (`X2` register). This requires a binary-wide analysis: we first found all `MetaClass` objects in all `InitFuncs` then identified the driver classes' inheritance relation if one class's `InitFunc` points to the `MetaClass` object of another class – its parent class (e.g., in the `X2` register).

We also analyzed I/O Kit, the kernel framework, using the above approach to recover classes `IOService` and `IOUserClient`, the ancestor classes of driver classes and *UserClients* (see Section 2).

3.2 Phase II: Discovering Driver Entry Points

Overview. Entry points are the interface functions that an Apple driver exposes to the user-space programs and are the starting points to build ICFG for an analysis. Based on the unique mechanisms Apple drivers use to interact with user-space programs we demystified and systematized in Section 2.3, we summarize two types of entry points for Apple driver analysis: **Type-1**) *user-entries* – virtual functions of *UserClients* (inherited from `IOUserClient`) that respond to user-space requests; and **Type-2**) *UserClient*-internal functions whose pointers are returned by *user-entries* (passed out of

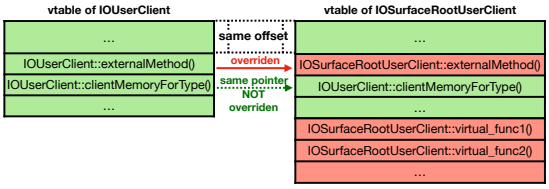

Figure 4: Example of vtable structure in class inheritance

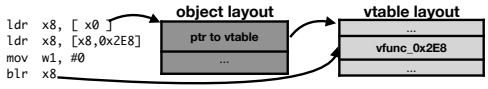

Figure 5: Illustration of virtual function calls

*UserClient*s, see Section 2.3). To find **Type-1** entry points, we look for the virtual functions in a UserClient's vtable that are inherited from `IOUserClient` (Table 2). **Type-II** entry points are organized by Apple in certain data structures in the data section of a driver binary for convenient driver management. We will show the first systematic approach to find the two types of entry points as follows.

Hunting for Type-I entry points. Our idea is to compare the vtable of a `UserClient` class and its parent `IOUserClient`. As Apple mandates single inheritance in C++, a virtual function in the parent's vtable is at the same offset with the child's vtable (see Figure 4). Since we have identified the vtables of `UserClients` in a binary and `IOUserClient` in Phase I, we can take the first few function pointers in `UserClient` vtable (up to the length of parent class vtable, see Figure 4) as potential *user-entries* we are looking for. Still, among these pointers some need to be filtered out. If a *user-entry* is not implemented (overridden) by `UserClient`, its invocation will flow into the parent's (`IOUserClient`) generic implementation, which (1) is in the kernel – invisible in the code space of driver analysis; and (2) has no effect – the generic implementation does not define what to do for specific subclasses. To filter out these, we compare the function pointers at the same offset between parent and child vtables: if the two function pointers are identical, that means the child class did not override the parent's virtual function – we filter out this *user-entry*.

Hunting for Type-II entry points. Those UserClient functions whose pointers are returned out by a *user-entry* (Type-II) are not directly referenced in the *user-entry* and returned, so a search in the *user-entry* code does not work. *Here we need to understand Apple's uniform management to these driver interfaces.* Specifically, for any *UserClient* function that will be exposed as an interface (except *Type-I*), Apple manages its function pointer and other information (e.g., input buffer size) in a data structure, called `method-struct`. All `method-structs` of a *UserClient* forms an array in the data section of a driver binary. Further, each time the *user-entry* returns one `method-struct` from the array (based on the `selector` argument in the system API call, see Figure 2); hence, our approach is to inspect the *user-entry* to find the reference to the array (i.e., find the loading of the array from data section), so as to find all function pointers it recorded.

At a lower level, *iDEA* takes a few steps: (1) find instructions in a *user-entry* where the referenced `method-struct` is finally stored in the return register (by convention `RAX` on macOS and `X0` on iOS, iPadOS, see implementation in Section 9); (2) use backward slicing [34, 81] to find instructions that affect the value in the return register; and (3) in these instructions, find an address-loading instruction that loads an address of `method-struct` array from data section. With this address, *iDEA* parses each element of the array – a `method-struct` – and finds the function pointers in it (see implementation details in Section 9).

3.3 Phase III: Identifying Objects with Vtables

Overview. Along the ICFG starting from a driver entry point, we observed many virtual function calls. As shown in Figure 5, a virtual function call on an object is an indirect call (e.g., through `BLR` instruction in arm64 and `call` in x86_64) whose target is retrieved from the vtable (a list of function pointers [35]) associated with the object's class. To resolve virtual-function calls, we need to identify the vtable assigned to the object during its instantiation.

For this purpose, prior approaches [31–33, 59, 67] rely on constructor-based analysis and specific code patterns of vtable assignment to identify vtables for constructed objects. However, those approaches cannot work on Apple platforms since the drivers classes are commonly instantiated through a generic kernel API (`allocClassWithName()`) without constructors, and the vtable assignment leverages a runtime feature (the above kernel API uses the runtime map of `MetaClass` objects to find vtable for the intended class based on class name and assign it to the object under construction, see Section 2).

In our research, since Phase I has identified vtables for each class, our approach is to infer the types (classes) of the objects when we traverse the ICFG, leveraging the understanding of runtime class instantiation and management in Apple drivers (which we systematized in Section 2). In particular, we analyzed its extensive usage of kernel API `OSMetaClass::allocClassWithName(char * classname)` for instantiating classes with names and objects passed as function arguments between driver classes, UserClients and Apple kernel. For a comprehensive analysis, we summarized three categories of objects to identify their types: (1) objects constructed locally in a driver function, e.g., through `allocClassWithName` (see Scenario 1, 5 below); (2) objects passed as arguments to driver functions (see Scenario 2, 3 below); and (3) objects returned by kernel APIs (see Scenario 4 below).

Scenario 1: objects instantiated through a classname. This is through the aforementioned kernel API `allocClassWithName(char * classname)`. We infer the type of the instantiated objects by analyzing the `classname` argument. This leverages known backward slicing and forward constant propagation techniques: backward slicing extracts the instructions affecting the `classname` argument, and forward constant propagation on the extracted instructions decides which string address is eventually passed to the argument. The discovered class name string indicates the object type.

Listing 1: `start()` of UserClient `IOReportUserClient`

```
bool IOReportUserClient::start(IOReportUserClient *this, IOService *a2) {
  v2 = OSMetaClasBase::safeMetaCast(a2, IOReportHub::MetaClassObj);
```

Scenario 2: driver class objects passed to `UseClient` as arguments. When a `UseClient` instance is created (through the `newUserClient() driver-callback`, see Figure 2), the kernel also

Figure 6: A part of a driver's configuration file

passed to it the driver instance. This is through calling `start()` on the `UseClient` instance, but the driver instance is passed in as a generic type `IOService *` (see Listing 1). The problem is, when the driver instance is referenced in the `UseClient` code (e.g., in a *user-entry*) to make virtual function calls, we don't know its type and vtable – which is essential to resolve the indirect calls. To solve the problem, we leverage Apple's driver registration and management, in particular, the information in the driver's configuration file and Apple's dynamic casting for driver classes:

• As mentioned in Section 2.1, Apple driver has a configuration file to register key information to the kernel (e.g., driver class and hardware type it can handle). The configuration file recorded a driver class' companion *UserClient* class to facilitate its management, as illustrated in Figure 6. Such information indicates the concrete type of driver instance passed to a particular UserClient such as IOReportUserClient.

• In occasional cases, the configuration information is not complete, which we complement through clue found in the code. Specifically, in `start()` function (Listing 1), we observed that `UserClient` often dynamically casts the generic-type (`IOService *`) driver instance to its concrete driver class type, through `OSMetaClassBase::safeMetaCast()` API. This function accepts an argument named "target type" (indicating casting target type), which is a pointer to a string or `MetaClass` object (a `MetaClass` object includes information about driver class name, see Section 2.2). Hence, we traced the reference of "target type" passed into the casting function, employing known backward slicing and forward constant propagation. This gives us the concrete type information for the driver instance.

Scenario 3: kernel objects passed to *driver-callbacks* as arguments. The driver callbacks (see Table 1) are virtual functions inherited from `IOService`, which is defined in the public kernel framework `I/O Kit` and comes with symbols, e.g., mangling function names [30]. Hence, we analyze the argument types of these callbacks through a conventional analysis on those mangling names [40]. For example, the mangling name of an `init` function in `IOService` is `__ZN9IOService4initEP12OSDictionary`, whose last partition indicates the argument type as `OSDictionary *`. Note that this is a one-off analysis step for all drivers.

Scenario 4: objects are return values or arguments of kernel API calls. Drivers often need to invoke kernel APIs to process data or create objects. The declaration of these kernel API can be found in Apple API documentation [9] or header files [7]. Based on the declaration, if the object to identify is the return value or used as an argument of kernel API call, we can infer its type.

Scenario 5: objects instantiated through constructors. Sometimes, driver objects are instantiated through constructors, for which we leverage traditional, constructor-based analysis to identify vtable installed into the starting address of the object [31–33, 59, 67]. Specifically, we recognize a constructor in the code based on certain code patterns, and Apple drivers' patterns are slightly different from other platforms: Apple driver constructors are created by compilers (based on fixed macros [4]) and, thus, show a few fixed patterns (we detail two common code patterns in Section 9).

Type propagation. Once we identified the types of certain objects (called *type sources*), we performed type propagation to infer the types of more objects. Along the control flow, type propagation passes the type of a *type source* to other objects if they are copied from the *type source*. In our approach, we keep an object-type map (τ) recording the confirmed types of objects (objects are referenced by registers and memory address at instruction level). We start our propagation from a basic block, by examining whether an object in the type set is copied to another object (by instructions such as `MOV` or `STR`). If true, we update the type of the destination object. Inversely, if we find an object in the map is assigned with a new value, we remove the object from the map. If a type-known object is stored in a memory region of another type-known object, we know that the latter object (and its class) has a member whose type is the former's type. After type propagation in a basic block, we carry the type set (τ) and continue propagation in the subsequent basic block and functions. Sometimes, inter-procedural type propagation needs to pause and wait until indirect calls are resolved (see Phase IV). Once they are resolved, a new round of type propagation is performed. Algorithm 1 in Appendix outlines our approach.

3.4 Phase IV: Resolving Indirect Calls

With the output of previous phases – recovered classes, vtables, and object types – we can resolve indirect calls along the control flow and produces ICFGs. Specifically, we trace the `load` and `add` instructions on an object, and look up its vtable to find which function pointer is used as the call target (based on the offset in vtable indicated in the instructions). In this way, we resolve the targets of indirect calls, and add an edge from the call site to the target in the ICFG. Considering polymorphism, we consider the child-class' virtual functions of the resolved object as potential targets, and add edges from the call site to the child-class' functions. Therefore, some nodes in the call graph may have multiple out-edges.

Besides virtual function calls, we observed another Apple driver-typical indirect-call scenario to handle, to make the ICFG more complete. Specifically, driver classes pass their function pointers out to kernel (through calling certain kernel APIs), and then it is up to the kernel to make indirect calls through the function pointer [24]. Such kernel APIs include `IOCommandGate::runAction()` and `IOWorkLoop::runAction()`. When *iDEA* finds such kernel API calls in driver code, it performs backward slicing to trace the function pointers passed to the APIs. Then, instead of flowing the analysis into the kernel code space, *iDEA* opts for directly extending the call graph with an edge from the call site to the target of the function pointer.

Phase III and IV run iteratively. After an object's type is identified, Phase IV resolves indirect calls on the object. After a call target is identified, more objects' types can be analyzed and identified.

3.5 Supporting Pluggable Policy Checkers

iDEA supports pluggable security policy checkers to detect driver vulnerabilities along the ICFGs. A checker is implemented as a plugin with two call-back functions, (`pre_instr_checker` and `post_instr_checker`), which are invoked by *iDEA* before and after each instruction when traversing the ICFG. The checkers check

each instruction and can leverage the results from Phases I~IV, i.e., the ICFG and object types identified along the control flow.

Analysis clients. *iDEA* provides two analysis clients, taint analysis client (TAC) and symbolic execution client (SEC), whose capabilities can be leveraged by checkers. For taint tracking, a checker can specify taint sources (i.e., specific registers and memory areas at specific instructions) and *iDEA* performs taint tracking by employing an off-the-shelf dynamic taint tracking tool Triton [62, 65]. Triton performs taint tracking at register/memory area level by dynamically simulating the execution of arm64/x86_64 instructions without using specific hardware devices [61]; the taint results are updated as *iDEA* processes each instruction along the ICFG and are available to checkers. *iDEA* employs another client SEC that supports symbolic execution. Again, this is enabled by directly integrating Triton. A checker can specify the registers/memory areas, which will be assigned symbolic values using Triton's `TritonContext` module. Triton processes symbolic values on each instruction as *iDEA* traverses the ICFG; the results, i.e., symbolic expressions for data in the registers and memory areas and path-constraints, are available to the checker.

4 SECURITY POLICY CHECKERS

This section elaborates on two security policy checkers we have implemented to detect common types of driver bugs, race condition issues (Section 4.1) and out-of-bound (OOB) read/write (Section 4.3). Our checkers have led to the discovery of more than 30 zero-day security bugs in Apple drivers across 15 OS versions (see Section 5).

4.1 Race Condition in Apple Drivers

Race condition is a common risk that could lead to serious vulnerabilities, in which multiple threads/processes use the same resource simultaneously. We observed that race conditions tend to occur in Apple drivers, whose management *implicitly* treats any driver/UserClient instance as a shared resource without protection. Different user-space threads can use the same `UserClient` handle to trigger a *user-entry* on the same `UserClient` instance (see Section 2), whose internal states thus should have been protected with locks [15].

A motivating example. To illustrate the risk of race condition in the context of Apple drivers, we show a new vulnerability found by *iDEA*, which has been acknowledged by Apple with CVE assigned.

```
IOThunderboltFamilyUserClient::externalMethod (…)
                    ...
                    ▼
IOThunderboltFamilyUserClient::removeXDListener (…) {
    v7 = v2->member42_IOMemoryDescriptor;
    if ( v7 ) {  v7->release();  }
```

Figure 7: A security bug found by *iDEA* (we converted the assembly code to pseudo code for better readability)

This bug is found in the `IOThunderboltFamilyUserClient` class – a `UserClient` of macOS driver `IOThunderboltFamily`. Figure 7 outlines the vulnerable code: in the function `removeXDListener()` of the UserClient, which is called by `externalMethod()` (a *entry-point* of the UserClient reachable by user-space programs), the `release()` function is called on a member variable `v7` of the UserClient (of type `IOMemoryDescriptor *`). The code introduces a bug because the `release()` function frees the member variable without taking any lock or concurrency protection. The security risk comes when two user-space threads simultaneously trigger this code. One thread may cause the member variable to be freed first, and, thus, the other thread will call `release()` on an already freed object, leading to a use-after-free (a.k.a., UAF [76]) vulnerability.

4.2 Race Condition Checker

We developed a security policy checker to detect race-condition risks. As a first step, we focused on two common types of bugs that race conditions can lead to – UAF and Null pointer dereference. Our checker, starting from driver entry-points, inspects each instruction, including direct and indirect calls, memory load, memory store, etc., with respect to a set of security policies, defined as follows.

Terminology. Before defining the security policies, we first define two types of terms, *operation* (*op*) and *function* (*func*).

- `op_nullify(member`$_A$`)`: an operation to store 0 into member$_A$.
- `op_release(member`$_A$`)`: a call to member$_A$'s `release()` function. On Apple platform, `release()` decreases the object's reference count by 1 [21]. When the reference count becomes zero, the referenced memory block is automatically freed.
- `op_retain`(member$_A$): a call to member$_A$'s `retain()` function. `retain()` increases the object's reference count by 1 [23].
- `op_use`(member$_A$): a call to member$_A$'s virtual function.
- `op_lock`: a call to lock-related kernel function, e.g. `IOLockLock` [15].
- `op_unlock`: a call to unlock-related kernel function, e.g. `IOLock Unlock` [15].
- `func_pathTo(op)`: the checker function to extract the instructions along the ICFG from entry-point to the `op`, yielding a path.
- `func_numberOf(op)`: the checker function to count the occurrences of `op`, given a path `p`.

Intuitively, the `ops` can be observed in driver code by the checker; the `funcs` are detection-related functions employed by the checker.

Security policies. A `release()` on a member variable (of a UserClient or driver class) should always be paired with (come after) a `retain()` and should happen by first acquiring a lock. Otherwise, a thread might reduce its reference count to zero and, thus, get it freed while another thread is using the member variable, leading to UAF. Similarly, setting a member variable to NULL should get a lock first. Otherwise, a Null pointer dereference on the member variable could happen in another thread. We define the security policies as follows.

- **Unsafe Release**: on path `func_pathTo(op_release(member`$_A$`))`, `func_numberOf(op_lock) == func_numberOf(op_unlock)` and `func_numberOf(op_retain(member`$_A$`))` == 0.
- **Unsafe Nullify**: on path `func_pathTo(op_nullify(member`$_A$`))`, `func_numberOf(op_lock) == func_numberOf(op_unlock)`.

Supplemental rules. For better detection accuracy, *iDEA* employs a set of rules to supplement the security policies as follows.

- To report an **Unsafe Release** or **Unsafe Nullify** bug, there should be at least one *op_use* on the same member variable through any entry-point of the driver – this is to account for the racing condition where a second thread *uses* the variable after the first thread releases or nullifies it.
- To report an **Unsafe Nullify** bug, the member variable in question should be a pointer to a class instance, not a constant.
- If an **Unsafe Release** is on an object passed into drivers by the kernel (e.g., through `UserClient`'s `start()`, see Listing 1), we do

not raise alarms for this object. This is because the object is created by the kernel, and we do not know its reference count unless we perform a thorough analysis of the kernel.

4.3 OOB Read/Write Checker

Out-of-bound (OOB) read and write is a common security risk [77] that also tends to happen in Apple drivers and can lead to arbitrary code execution in kernel space, or kernel data leakage and corruption. This happens when drivers take user-space inputs as indexes to access buffers in the kernel space without boundary checks.

A motivating example. To illustrate the risk of OOB in Apple drivers, we showcase a new vulnerability found by *iDEA*, that enabled a malicious user-space program to get kernel privilege. The bug was just reported to Apple, which is fixing it and asked us to keep its details confidential (so we omit its driver name and vulnerable function name in its description). In this bug, the user space interacts with a *UserClient* through its `getTargetAndMethodForIndex()` *user-entry* (see Table 2). The *UserClient* performed the following without a boundary check: (1) retrieve a 32-bit integer from the buffer in user-space input (an argument of system API call, see Figure 2); (2) add the integer to a kernel buffer pointer without checking the integer value; and (3) read from and write to the kernel buffer using the pointer, which can be out of boundary. Also, the data to write comes from the user-space buffer.

As a result, this vulnerability allows a malicious user-space program to write crafted data beyond the intended kernel buffer, corrupting other critical kernel data near the buffer. We implemented an end-to-end full-chain exploit by combining known exploitation techniques including heap spray [36] and heap feng-shui [71]; it successfully achieved arbitrary code execution in the kernel, and completely controlled a macOS 10.15.4 (the latest version when we reported the vulnerability) system (see attack demo online [2]).

OOB security policy. The high-level policy for OOB detection is simple: the driver accesses a buffer in the kernel space using an index from user-space inputs without boundary check. Note that user-space inputs come from the arguments of entry-points and buffers storing user inputs (e.g., `scalarInput` in Figure 2).

OOB checker. To apply the policy for OOB bug detection, our implemented checker leverages the taint analysis enabled by the *iDEA* framework. From a high-level, the checker specifies the user(-space) inputs as taint sources, and *iDEA* will perform taint tracking. Before and and after each instruction along the ICFG, call-back functions of the checker will be invoked by *iDEA*, to determine whether any tainted value is used to access memory buffer. The dereferenced access is identified at instruction level by examining memory load and store instructions (e.g., `LDR`, `STR`, `mov`). If the access through a tainted pointer indeed happened, the checker then checks whether there is a condition check on the original tainted user-input data along the control-flow path. The lack of condition checks indicates an OOB bug.

The key low-level details relate to two aspects:

• The taint sources are set to arguments of driver entry-points and data in user-input buffers, e.g., `scalarInput` in system API call (Figure 2). Each separate argument or small chunk in a buffer (4 bytes as a chunk in our implementation) is assigned a unique taint.

• To check the existence of condition check on a tainted data, *iDEA* employs its symbolic execution client to assign a symbol to its taint source, and resolves constraints if any on the symbol along the ICFG. No constraint, i.e., intuitively no checks on its value, indicates an OOB bug regarding the taint source.

5 EVALUATION

In this section, we evaluate the effectiveness and performance of *iDEA*. We ran *iDEA* over 3,400 driver binaries across 15 OS versions to evaluate the overall effectiveness, including bug findings, false positives, etc. (Section 5.1). Note that this is the first known large-scale security analysis on Apple drivers. We also evaluate individual analysis phases of *iDEA* (Section 5.2) and the overall performance overhead (Section 10 in Appendix), and further compare *iDEA* with state-of-the-art works for C++ binary analysis (Section 5.3). The comparison showed that prior works are generally ineffective for analyzing Apple drivers. We will discuss the limitations of *iDEA* in Section 6.

5.1 Evaluating Overall Effectiveness

Driver set. We used 3,400 driver binaries from 15 OS versions, including iOS (8, 9, 10, 11, 12, 13, 13.6.1 – the latest), macOS (10.13, 10.14, 10.15, 10.15.6 – the latest), iPadOS (13.1, 13.6.1 – the latest), and tvOS (13.2, 13.4.8 – the latest). On Apple platforms, a certain driver binary may be used on different OSes: e.g., a driver binary named `IOSurface` is found on iOS, tvOS, and iPadOS, and also on different versions of these OSes; still, its implementation can be different on each OS and version. Without loss of generality, we count each driver binary on a specific OS version, yielding a total of 3,400 driver binaries. The driver binaries were obtained from the OS update packages available online [46].

Bug findings. Altogether, *iDEA* reported 50 unique bugs in all 3,400 driver binaries. Note that if two bugs are found on two OS versions but with the same driver name, policy violation, and (call-graph) path to the bug, we consider them as one bug. We manually confirmed 46 are true positives, which include 13 UAF, 28 NULL pointer dereferences, and 5 OOB bugs, spanning macOS, iOS, iPadOS, and tvOS (see all vulnerabilities in Table 6 in Appendix, including their OSes, driver names, vulnerability types, etc.). Among the 46 true positives, 35 are zero-day vulnerabilities; the other 11 are publicly unknown vulnerabilities in older OS versions (e.g., iOS 8-12, macOS 13-14), which have been silently fixed in later OS versions by Apple. We have reported all 35 zero-day vulnerabilities to Apple: five vulnerabilities have been assigned CVEs with Apple acknowledgement; two are acknowledged and CVE assignments are scheduled; others are still going through Apple's internal investigation process (Apple requested us to keep them confidential).

Bug confirmation with PoC exploits. We manually confirmed all 35 zero-day bugs found by *iDEA*. This combines PoC exploits on real Apple devices we own (see device list below), and manual inspection of vulnerable code. To this end, we developed proof-of-concept user-space programs to trigger the bugs on corresponding OSes: a system crash indicates a successful exploit. To confirm a bug, we further confirm that the crash is caused by our exploit: the crash indeed comes with kernel panic log (under `/Library/Logs/DiagnosticReports/`), and the stack trace in the log shows the crash is

caused by our target driver and its vulnerable function. We developed PoC exploits for 10 zero-day bugs, which all caused system crash and were confirmed. We show the source code of one PoC in Listing 2 in Appendix. The PoC exploits were done on our Apple computers/devices, including a MacBook Air, MacBook Pro, iMac mini, iPad mini 2, and iPhone 6s, XR and 11 pro.

For the other 25 bugs, we cannot actually run and exploit their drivers since we don't have the corresponding hardware devices. For example, the FireWireAudio driver requires an audio device [56] connected through a firewire [12] cable; AppleIntel8254XEthernet driver must be run on Mac Pro which we don't have. For these bugs, we disassembled the binary and manually confirmed them by comparing the vulnerable code with those we have confirmed through PoC. This worked since vulnerable drivers often made similar mistakes, e.g., nullifying a member variable in `clientClose()` (a *user-entry* to close `UserClient` when `IOServiceClose()` system API is called) without taking any locks.

Serious impacts. Besides system crash, the bugs we found can enable a malicious user-space program to gain kernel privilege and run arbitrary code in the kernel space. We developed an end-to-end exploit on macOS 10.15.4 (the latest at the time of the finding), with a video demo online [2].

False positives. There are four false positives in our findings. The root cause of these false positives is that our current security check is flow-sensitive but not context-sensitive. For example, in a macOS *user-entry*, i.e., `AHCISMARTUserClient::externalMethod()`, there is a `release` operation on a member variable without lock-protection, for which our checker raised a false alarm. After a manual analysis, we found that the `release` operation is under a branch which is restricted through a condition check. Based on the condition check, the `release` operation is only reachable through a path where a previous `retain` operation is performed (`retain` increases the reference counter of the member variable, a common mechanism to prevent UAF [76]); hence, the `release` is always safe. The false positive occurs because our current analysis is not context-sensitive, i.e., we do not resolve the condition check and then simply opt for both branches to build control-flow graphs when a condition check is met.

5.2 Evaluating Individual Analysis Phases

We also evaluated the effectiveness of key individual phases (see Section 3) of *iDEA* using multiple Apple OSes. To this end, we ran *iDEA* over all driver binaries on the latest macOS 10.15.6 (362 driver binaries), iOS 13.6.1 (176 driver binaries), iPadOS 13.6.1 (189 driver binaries), and tvOS 13.4.8 (143 driver binaries).

Recovering classes. In the evaluation on multiple OSes, *only* macOS drivers come with debug symbols that can provide ground truth for recovering classes. We will first show the results with macOS, and then *all* other OSes.

For all 8,217 classes (including 3,841 `MetaClasses` generated by the driver compiler) in all the 362 driver binaries, *iDEA* accurately recovered 7,666 (93%) classes, including their vtables, inheritance hierarchies, class names, and class sizes. The approach of *iDEA* handles driver classes, `UserClients`, `MetaClasses`, and all classes inherited from classes in Apple's kernel programming framework (e.g., I/O Kit, the driver programming framework); for these "Apple-rooted" classes (accounting for 93% of all classes in the driver binaries), *iDEA* has 100% precision in class recovery. This is because *iDEA* leverages a reliable information source, unlike heuristics in prior works (see Section 3.1). The 7% that *iDEA* did not recover are utility classes implemented by driver developers (not "rooted" from Apple framework and do not come with InitFuncs); they may be handled by prior approaches in recovering C++ classes, such as [59, 67], which the current implementation of *iDEA* did not include.

On iOS, iPadOS and tvOS (see OS versions above), *iDEA* recovered 3,536, 3,848, and 3,274 classes, respectively. We lack ground truth as Apple drivers on these platforms are generally closed-source and without debug symbols. Still, we found Apple released the source code of four driver binaries [14, 16–18] on the three OSes (see driver names in Table 5 in Appendix), and used their class information as ground truth. For the four drivers, *iDEA* recovered all 140 out of 140 (100%) classes for each OS with 100% precision.

Discovering driver entry points. On macOS, iOS, iPadOS, and tvOS (see OS versions above), *iDEA* found 1,426, 348, 371, and 310 entry points, respectively. Again, the four drivers with source code provided us ground truth. For the four drivers, *iDEA* successfully found *all* 54 entry points with 100% precision on each OS.

Resolving indirect calls. On macOS, *iDEA* resolved 139,282 out of 209,558 indirect calls (66%) found in the 362 driver binaries. Similarly, on iOS, iPadOS, and tvOS, *iDEA* resolved 63% (58,163/92,464), and 63% (66,861/105,950), 59% (43,621/74,768) of indirect calls, respectively. Unlike prior works such as [59] that can generate ground truth at compile time, we were not able to compile the four drivers even with source code due to the lack of Apple-internal driver SDK (with errors such as "cannot find sdk iphoneos.internal" in Xcode [13]).

5.3 Comparison with Prior Works

We also compared *iDEA* with state-of-the-art techniques/tools (for C++ binary analysis) in analyzing Apple drivers. In particular, we ran *Marx* [59] and *iDEA* over all 362 driver binaries on the latest macOS 10.15.6, and compared the results for recovering classes and resolving indirect calls. We show that *iDEA* significantly outperformed *Marx* on Apple platforms and discuss the reasons.

Experiment preparation. The experiment was done after we carefully made *Marx* Apple-aware, i.e., capable of handling Apple's Mach-O [83] binary format. In its original design, *Marx* could analyze x86_64 binaries (in ELF and PE format [82, 84]) on Linux and Windows; for this purpose, *Marx* must first extract information (functions, data, vtables, symbols, etc.) from the corresponding sections of a binary (the sections were prepared in IDA pro). Since Mach-O has different sections and section layouts to hold the information, what we need to do is to retrofit the implementation of *Marx*, so it is aware of Mach-O sections and what information (e.g., functions, data, etc.) could be found in which sections. We keep *Marx*'s algorithms intact, including those to identify vtables, infer class hierarchies, resolve indirect calls, etc. We released our Apple-aware version of *Marx* online [2].

Recovering classes. For a total of 8,217 classes in all the 362 driver binaries (macOS drivers have debugs symbols as ground truth, see

Section 5.2), *Marx* (Apple-aware) recovered 7,186 (87%) of all classes by finding their vtables. However, it incured a 34% false positive rate, i.e., *Marx* reported 10,963 vtables, among which 34% were not vtables. In comparison, on the same set of drivers, *iDEA* achieved a 100% precision and 93% coverage (see Section 5.2).

For recovering class hierarchies, *Marx* could not identify the direction of class inheritance, but only group classes that have the same ancestors into a hierarchy. However, in Apple's driver programming model (see Section 2), all classes have the same ancestor (`OSMetaClassBase` [19]), even for classes of functionality-unrelated drivers. This makes *Marx*'s results much less useful on Apple platforms. For example, *Marx* intends to use the recovered class hierarchies against control flow hijacking by ensuring virtual function calls conform to the class hierarchy [59]) (when all/most classes are in the same hierarchy, the defense is less useful). In contrast, *iDEA* could accurately identify the inheritance relation and direction using the reliable information source (see Section 3.1).

Resolving indirect calls. Among the 209,558 indirect calls in all 362 driver binaries, *Marx* only resolved 42,223 (20%) indirect calls, while iDEA resolved 66%. This indicates *iDEA* can build much more complete control-flow graphs in analyzing Apple drivers than *Marx*.

Discussion. As mentioned earlier (Section 1), prior works, including *Marx* [59], rely on constructor-based analysis and specific code patterns of vtable assignment to identify vtables for constructed objects (so as to resolve indirect calls on the objects). However, these techniques become much less effective on Apple platforms. In Apple drivers, classes are often instantiated through a generic kernel API `allocClassWithName()` (without constructors), which internally leverage the runtime map of `MetaClass` objects to assign vtables for the objects being instantiated (see Section 2). Without analyzing Apple's dynamic mechanism for class instantiation, prior works cannot properly identify vtables for objects, not to mention resolving indirect calls made on the objects.

Also, to find vtables, *Marx* relies on a few heuristics to match specific patterns in the data sections and patterns of references in the code section (e.g., only the beginning of the function entries is referenced from the code). Such heuristics are far less effective on Apple platforms, since we observed a substantial amount of vtable-like data structures (an array of function pointers) in Apple drivers. For example, the Type-2 *entry-points* are arrays of function pointers organized similarly to vtables (see Section 3.2) and were mistakenly identified as vtables by *Marx*, as found in our evaluation.

Other state-of-the-art works. We also attempted to compare *iDEA* with other state-of-the-art works [49, 67] for C++ binary analysis. We found it requires significant engineering efforts to make them Apple-aware due to incompatible disassemblers, different instruction sets, and call conventions, etc. For example, the disassembling framework relied on by [67] does not support Mach-O. This further indicates the lack of a tool like *iDEA* that can properly analyze Apple drivers.

6 DISCUSSION

Unique aspects of driver security on Apple platforms. Due to the unique paradigm of Apple driver programming and management (see Section 2), there are types of risks that Apple drivers are especially susceptible to, as indicated by our results. For example, Apple's management of driver instances can easily incur race conditions (see Section 4.2), a common problem uncovered by our large-scale study. This is because Apple kernel *implicitly* treats any driver/UserClient (class) instance as a shared resource: multiple, untrusted user-space threads/processes can operate on the same driver/UserClient instance (through system API calls, see Figure 2), whose internal operations (e.g., use/free of member variables) thus should generally have been protected, e.g., through locks. However, we found that Apple does not have a mechanism to enforce a concurrency protection (e.g., mandatory adding of locks), nor did Apple properly communicate the risks to driver developers, based on public documentations.

Such an observation will help derive security guidelines for Apple driver development, which are lacking today. Also, unlike the driver management on Linux, Windows, Android, etc., that has been well understood with their drivers extensively analyzed [26, 29, 50, 51, 57, 58, 63, 66, 68, 69, 74, 79, 87], our study brings to light that driver management on Apple is unique, opaque, and proprietary to Apple, posing a major obstacle for systematically understanding its security qualities and risks. Our research shows that a systematic analysis of Apple drivers requires an analysis of the driver management with driving programming model, registration, interactions with user-space programs, and the advanced runtime features to facilitate driver instantiation and usage. Note that the Apple driver management we demystified in Section 2 lacks public information, which we systematized through analyzing Apple I/O Kit (the kernel framework) and driver binaries.

Further, our tool – the first for automatic Apple driver analysis, can enable future research to systematically investigate the unique aspects/risks of Apple drivers and systems, a ubiquitous but far less explored ecosystem whose security quality can introduce serious implications.

Limitations of static analysis on Apple platforms. Although our static analysis achieved favorable results (e.g., tens of zero-day bugs found, significantly outperforming prior tools), a full analysis for the drivers on Apple platforms still faces challenges that require non-trivial new research, as observed in our study. In particular, Apple developed a set of advanced, runtime features to facilitate driver management and usage, which our current approach cannot fully handle (we handled a few runtime features, see Section 3). For example, the kernel organizes instances of all drivers in the system in a runtime pool. An Apple driver can call a kernel API `IOService::waitForMatchingService()` [25] to retrieve a driver (instance) from the pool that matches specified criteria, such as a category, name, or ID [10], and then calls its virtual functions. For instance, the iPhone camera driver would retrieve an instance of light sensor driver available on the phone, and uses its functionalities when taking pictures. Such a runtime feature makes the virtual function calls hard to resolve using our current approach, since we may not know the types (i.e., classes) and vtables of the retrieved driver instance, which is selected by the kernel at runtime and dependent on the drivers available on the specific system (e.g., an iPhone/iPad/iMac of a specific model). To handle the situation, one needs to analyze all drivers available on the particular system and how Apple kernel selects the driver instance. Besides, Apple provides other kernel APIs, such as `resourceMatching()` [22], `fromPath()` [11],

`propertyMatching()` [20], that feature advanced, runtime driver management. *A systematic analysis of those Apple-unique features that can build upon the analysis capabilities offered by iDEA will be left to our future research.*

Limitations of *iDEA* and false negatives. False negatives can be introduced due to the limitations mentioned above and a few other limitations of *iDEA*. First, *iDEA* currently does not analyze control-flow across binaries: if a driver imports classes from another driver binary, *iDEA* can not resolve indirect calls on objects instantiated from the imported classes. For example, a driver of a Bluetooth keyboard would call certain utility functions implemented in a generic Bluetooth driver. To handle this situation, one needs to figure out the drivers' dependency tree, identify calls to external functions, and share analysis results (e.g., class recovery) across binaries. Second, our current OOB checker is relatively simple, which only checks the (non-)existence of constraint (for tainted values used to access memory buffers), instead of solving the constraint. This may lead to false negatives. Solving the constraint and enabling a more precise boundary check will be left to our future work by improving the OOB checker, which is pluggable to *iDEA*.

Differences and homogeneity between Apple OSes. For the ease of driver management and programming, Apple OSes share not only the management of drivers (see Section 2), but even source code of a portion of drivers. We found that 46 drivers are shared by the latest macOS, iOS, iPadOS, and tvOS (see OS versions in Section 5.2), with the same bundle IDs [10] (Apple identifier of Mach-O binaries) and code. The major differences across Apple OSes are the instruction sets, since macOS drivers use x86_64 instructions while iOS/iPadOS/tvOS uses arm64 instructions, which we have to process differently in our implementation.

7 RELATED WORK

Security analysis on Apple drivers. As mentioned earlier (Section 1), prior works performed analysis on iOS [1, 27, 36, 52, 78, 85, 86] or macOS drivers [28, 41, 53, 66, 70], which, however, heavily relied on manual reverse engineering efforts and OS-specific heuristics. *iDEA* is automatic in bug finding generally across major Apple OSes. On iOS, for example, [1, 36, 52, 72, 73, 78, 86] showed how they manually discovered vulnerabilities and exploited them in iOS/macOS drivers and the kernel. The most related to *iDEA* are [27, 85], which applied to iOS and macOS. However, their bug finding is manual; these tools were mainly designed to assist manual bug finding by complementing IDA pro [42] (using class information recovered from binaries). Still, their class recovery did not employ a general approach in the sense that, on iOS and macOS they leveraged different heuristics, e.g., using symbols only available in macOS drivers [85]. On macOS, prior driver security analysis [28, 41, 53, 66, 70] are mostly based on dynamic approaches. For example, kAFL [66] and LynxFuzzer [53] utilize hardware-assisted virtualization to fuzz macOS drivers. PassiveFuzzFrameworkOSX [70] rewrites the kernel code at runtime to fuzz macOS drivers. Their approaches cannot work for other Apple OSes (iOS, tvOS, iPadOS, etc.), since they are more restricted than macOS: they do not allow kernel instrumentation due to runtime integrity check [45]; hardware-assisted virtualization used in these approaches is unavailable on these customized Apple devices.

Security analysis on drivers of other platforms. Prior works [26, 29, 50, 51, 57, 58, 63, 66, 68, 69, 74, 79, 87] studied driver security on other platforms including Linux, Window, and Android. On Linux, Dr. Checker [51] is a fully-automated static analysis tool that uses pointer and taint analysis to find general bugs in driver source code in C language. SymDriver [63], S2E [29] and WatSym [58] employ symbolic execution for verifying properties and finding vulnerabilities on Linux drivers. On Android, Charm [74] facilitates dynamic analysis of Android device drivers by exporting Android drivers to a virtual machine on a workstation. ADDICTED [87] aims at detecting flaws in customized Android drivers. On Windows, kAFL [66] and Digtool [57] fuzz Windows drivers using hardware-assisted virtualization and execution tracing techniques. [50] performs static analysis on Windows drivers' binaries to verify API specifications on drivers. Besides examining drivers' binary code, Microsoft provides a tool called SDV [26] to analyze driver source code. Driver analysis techniques on non-Apple platforms cannot be applied for Apple due to the Apple-unique driver management (e.g., unique entry points, driver regisgtration and dynamic instantiation), programming model, and even availability of source code. For example, driver analysis on Linux can leverage their C-language source code, while Apple drivers are closed-source and programmed in C++.

Static analysis on C++ binaries. Many works, such as [31–33, 49, 59, 67], have been proposed for the analysis of C++ binaries. For example, [32, 48, 49, 59] depend on heuristics and specific code patterns to identify vtables in data sections, their references in code sections, and class hierarchies. Prior works [31–33] also rely on heuristics to identify constructors, rely on code features for constructors and vtables assignment to resolve indirect calls. Some other works [38, 39, 64] relied on RTTI data structures embedded in unstripped C++ binaries to facilitate reverse engineering of C++ binaries. However, such RTTI information is absent in Apple drivers. As evaluated and discussed in Section 5.3, prior works, including [49, 59, 67], typically rely on constructor-based analysis, specific code patterns, or heuristics to recover classes, identify vtables, and resolve indirect calls. However, Apple leverages unique runtime features to manage classes (e.g., class instantiation using a name and vtable assignment using a runtime map), with constructors removed and many vtable-like data structures, which make prior works ineffective on Apple platforms.

8 CONCLUSIONS

In this paper, we propose *iDEA*, an automatic static analysis platform for checking the security of Apple drivers. We systematically identify the unique challenges in automatic static analysis of Apple drivers and tackle the challenges with new, Apple-general techniques. Based on *iDEA*, we customize some security policies to automatically detect UAF and Null-Pointer dereference vulnerabilities caused by race condition and out-of-bound read/write. We ran *iDEA* to analyze 3,400 driver binaries of 15 Apple OS versions, which resulted in the discovery of more than 40 previously unknown bugs. *iDEA* incurs a low false positive rate and time overhead.

REFERENCES

[1] Adam Donenfeld. 2018. Viewer Discretion Advised: (De)coding an iOS Kernel Vulnerability. http://phrack.org/papers/viewer_discretion_advised.html.

[2] anonymous author. 2020. iDEA Supporting Website. https://sites.google.com/view/idea-apple-driver.

[3] Apple Inc. 2004. The libkern Base Classes. (2004). http://mirror.informatimago.com/next/developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/BaseClasses/chapter_6_section_2.html.

[4] Apple Inc. 2014. The Base Classes. https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/BaseClasses/BaseClasses.html.

[5] Apple Inc. 2014. Introduction to I/O Kit Fundamentals. https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/Introduction.html.

[6] Apple Inc. 2018. About Information Property List Files. https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html.

[7] Apple Inc. 2018. XNU source code. (2018). https://opensource.apple.com/source/xnu/.

[8] Apple Inc. 2019. Apple security updates. (2019). https://support.apple.com/en-us/HT201222.

[9] Apple Inc. 2019. Kernel. https://developer.apple.com/documentation/kernel.

[10] Apple Inc. 2020. CFBundleIdentifier. https://developer.apple.com/documentation/bundleresources/information_property_list/cfbundleidentifier.

[11] Apple Inc. 2020. fromPath(const char *, const IORegistryPlane *, char *, int *). https://developer.apple.com/documentation/kernel/ioregistryentry/1810742-frompath.

[12] Apple Inc. 2020. Identify the ports on your Mac. https://support.apple.com/en-us/HT201736.

[13] Apple Inc. 2020. Introducing Xcode 12. https://developer.apple.com/xcode/.

[14] Apple Inc. 2020. IOHIDFamily-1446.80.2. https://opensource.apple.com/source/IOHIDFamily/IOHIDFamily-1446.80.2/.

[15] Apple Inc. 2020. IOLocks.h. https://opensource.apple.com/source/xnu/xnu-6153.61.1/iokit/IOKit/IOLocks.h.auto.html.

[16] Apple Inc. 2020. IONetworkingFamily-139.60.1. https://opensource.apple.com/source/IONetworkingFamily/IONetworkingFamily-139.60.1/.

[17] Apple Inc. 2020. IOPCIFamily-370.81.1. https://opensource.apple.com/source/IOPCIFamily/IOPCIFamily-370.81.1/.

[18] Apple Inc. 2020. IOStorageFamily-238.0.1. https://opensource.apple.com/source/IOStorageFamily/IOStorageFamily-238.0.1/.

[19] Apple Inc. 2020. OSMetaClassBase. https://developer.apple.com/documentation/kernel/osmetaclassbase.

[20] Apple Inc. 2020. propertyMatching. https://developer.apple.com/documentation/kernel/ioservice/1810622-propertymatching.

[21] Apple Inc. 2020. release. https://developer.apple.com/documentation/kernel/osobject/1941151-release.

[22] Apple Inc. 2020. resourceMatching(const char *, OSDictionary *). https://developer.apple.com/documentation/kernel/ioservice/1810840-resourcematching.

[23] Apple Inc. 2020. retain. https://developer.apple.com/documentation/kernel/osobject/1941154-retain.

[24] Apple Inc. 2020. runAction. https://developer.apple.com/documentation/kernel/iocommandgate/1811576-runaction.

[25] Apple Inc. 2020. waitForMatchingService. https://developer.apple.com/documentation/kernel/ioservice/1811164-waitformatchingservice.

[26] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. 2006. Thorough static analysis of device drivers. ACM SIGOPS Operating Systems Review 40, 4 (2006), 73–85.

[27] bazad. 2018. ida_kernelcache: An IDA Toolkit for analyzing iOS kernelcaches. https://github.com/bazad/ida_kernelcache.

[28] Ian Beer. 2014. pwn4fun Spring 2014–Safari–Part II. (2014). https://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html.

[29] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. ACM Sigplan Notices 46, 3 (2011), 265–278.

[30] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. 2019. 5.1 External Names (a.k.a. Mangling). https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

[31] David Dewey and Jonathon T Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code.. In NDSS.

[32] David Dewey, Bradley Reaves, and Patrick Traynor. 2015. Uncovering Use-After-Free Conditions in Compiled Code. In Availability, Reliability and Security (ARES), 2015 10th International Conference on. IEEE, 90–99.

[33] David Bryan Dewey. 2015. Finding and remedying high-level security issues in binary code. Ph.D. Dissertation. Georgia Institute of Technology.

[34] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications.. In NDSS. 177–183.

[35] Margaret A Ellis and Bjarne Stroustrup. 1990. The annotated C++ reference manual. Addison-Wesley.

[36] Esser, Stefan. 2011. Exploiting the iOS kernel. Black Hat USA (2011).

[37] Flanker. 2016. The Python Bites your Apple Fuzzing and exploiting OSX Kernel bugs. https://papers.put.as/papers/macosx/2016/xkungfoo.pdf.

[38] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: approaching C++ decompilation. In 2011 18th Working Conference on Reverse Engineering. IEEE, 347–356.

[39] Alexander Fokin, Katerina Troshina, and Alexander Chernov. 2010. Reconstruction of class hierarchies for decompilation of C++ programs. In Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on. IEEE, 240–243.

[40] GNU. 2020. Demangling. https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html.

[41] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2345–2358.

[42] Hex-Rays. 2015. IDA: About. (2015). https://www.hex-rays.com/products/ida/.

[43] Ian Beer. 2018. CVE-2017-13861. https://bugs.chromium.org/p/project-zero/issues/detail?id=1417.

[44] Ian Beer. 2019. A very deep dive into iOS Exploit chains found in the wild. https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html.

[45] iOS Expert. 2017. Apple's iOS Kernel Patch Protection (KPP) Explained. (2017). https://yalujailbreak.net/kernel-patch-protection/.

[46] Just a Penguin. 2019. IPSW Downloads. (2019). https://ipsw.me/.

[47] Kaspersky Lab. 2017. Pegasus: The ultimate spyware for iOS and Android. (2017). https://www.kaspersky.com/blog/pegasus-spyware/14604/.

[48] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating types in binaries using predictive modeling. In ACM SIGPLAN Notices, Vol. 51. ACM, 313–326.

[49] Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. Statistical reconstruction of class hierarchies in binaries. In ACM SIGPLAN Notices, Vol. 53. ACM, 363–376.

[50] Johannes Kinder and Helmut Veith. 2010. Precise static analysis of untrusted driver binaries. In Formal Methods in Computer-Aided Design (FMCAD), 2010. IEEE, 43–50.

[51] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, 1007–1024.

[52] Mandt, Tarjei. 2013. Attacking the iOS Kernel: A Look at'evasi0n'. http://www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf.

[53] Stefano Bianchi Mazzone, Mattia Pagnozzi, Aristide Fattori, Alessandro Reina, Andrea Lanzi, and Danilo Bruschi. 2014. Improving mac os x security through gray box fuzzing technique. In Proceedings of the Seventh European Workshop on System Security. ACM, 2.

[54] Micorsoft. 2019. WDF_DRIVER_CONFIG structure. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdfdriver/ns-wdfdriver-_wdf_driver_config.

[55] Micorsoft. 2019. WdfDriverCreate function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdfdriver/nf-wdfdriver-wdfdrivercreate.

[56] Music Matter. 2015. 6 Of The Best Firewire Audio Interfaces 2015. https://www.musicmatter.co.uk/lists/best-firewire-audio-interfaces-2015.

[57] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, 149–165.

[58] Riyad Parvez, Paul AS Ward, and Vijay Ganesh. 2016. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering. IBM Corp., 116–127.

[59] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs.. In NDSS.

[60] Plaskett, Alex and Loureiro, James. 2017. Biting the Apple that feeds you - macOS Kernel Fuzzing. https://labs.f-secure.com/archive/biting-the-apple-that-feeds-you-macos-kernel-fuzzing/.

[61] Quarkslab. 2019. Taint analysis on aarch64 binaries? https://github.com/JonathanSalwan/Triton/issues/837.

[62] Quarkslab. 2020. Triton - A DBA Framework. https://triton.quarkslab.com/.

[63] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. 2012. SymDrive: Testing Drivers without Devices. In Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). USENIX, Hollywood, CA, 279–292. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/renzelmann.

[64] Paul Vincent Sabanal and Mark Vincent Yason. 2007. Reversing C++. (2007). https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf.

[65] Florent Saudel and Jonathan Salwan. 2015. Triton: A dynamic symbolic execution framework. In Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes. 31–54.

[66] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels.

In 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.
[67] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. 2018. Using logic programming to recover C++ classes and methods from compiled executables. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 426–441.
[68] Semmle. 2020. Semmle. https://semmle.com/.
[69] ShiftLeftSecurity. 2020. Joern Documentation. https://joern.io/docs/.
[70] SilverMoonSecurity. 2016. PassiveFuzzFrameworkOSX. (2016). https://github.com/SilverMoonSecurity/PassiveFuzzFrameworkOSX.
[71] Alexander Sotirov. 2007. Heap feng shui in javascript. Black Hat Europe 2007 (2007).
[72] Stefan Esser. 2011. IDA-IOS-Toolkit. https://github.com/stefanesser/IDA-IOS-Toolkit.
[73] Stefan Esser. 2011. Targeting the iOS Kernel. https://papers.put.as/papers/ios/2011/SysScan-Singapore-Targeting_The_IOS_Kernel.pdf.
[74] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: facilitating dynamic analysis of device drivers of mobile systems. In 27th {USENIX} Security Symposium ({USENIX} Security 18). 291–307.
[75] The Clang Team. 2019. LibClang. https://clang.llvm.org/docs/Tooling.html.
[76] The MITRE Corporation. 2019. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.
[77] The MITRE Corporation. 2020. CWE-787: Out-of-bounds Write. https://cwe.mitre.org/data/definitions/787.html.
[78] Tielei Wang, Hao Xu, and Xiaobo Chen. 2016. Pangu 9 internals. Black Hat USA (2016).
[79] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). USENIX, Hollywood, CA, 163–177. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang.
[80] Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS) 13, 2 (1991), 181–210.
[81] Mark Weiser. 1984. Program slicing. IEEE Transactions on software engineering 4 (1984), 352–357.
[82] Wikipedia. 2020. Executable and Linkable Format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.
[83] Wikipedia. 2020. Mach-O. https://en.wikipedia.org/wiki/Mach-O.
[84] Wikipedia. 2020. Portable Executable. https://en.wikipedia.org/wiki/Portable_Executable.
[85] Xiaolong Bai, Min (Spark) Zheng. 2018. Eating The Core of an Apple: How to Analyze and Find Bugs in MacOS and iOS Kernel Drivers. https://conference.hitb.org/hitbsecconf2018ams/sessions/eating-the-core-of-an-apple-how-to-analyze-and-find-bugs-in-macos-and-ios-kernel-drivers/.
[86] Zhenquan Xu, Gongshen Liu, Tielei Wang, and Hao Xu. 2017. Exploitations of uninitialized uses on macos sierra. In 11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17).
[87] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The peril of fragmentation: Security hazards in android device driver customizations. In Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 409–423.

APPENDIX

9 IMPLEMENTATION

We implemented *iDEA* as an IDA pro [42] plugin (15,000 lines of Python code), which performs analysis on the assembly code of driver binaries. We will release the source code. In the following, we provide implementation details for *iDEA*, with respect to key phases in its design (Section 3).

Recovering classes. For recovering classes, we implement the backward slicing [34, 81] and forward constant propagation algorithm [80] on both arm64 instructions (for iOS/iPadOS/tvOS) and x86_64 instructions (for macOS). We inspect registers that store addresses of `MetaClass` object, class name string, and class size during the analysis of *InitFunc*s. For finding driver class vtable in analyzing `MetaClass::alloc()`, we identify the invocation of `new()` allocation function, and perform forward data flow analysis on the newly-allocated object (whose address is stored in register `X0` for arm64 or `RAX` for x86_64) and look for the code pattern that stores an data section address (i.e., vtable address) into the starting address (e.g., `[X0]` or `[RAX]`) of the allocated object .

Besides recovering classes, *iDEA* in Phase I also complements IDA pro's auto-analysis on the driver binaries. Specifically, IDA pro can not properly recognize many function bodies in iOS/iPadOS/tvOS drivers. We recognize functions by looking for stack-push (e.g. `STP X29, X30,[SP,-0x10]!`) and stack-pop (e.g. `LDP X29,X30,[SP],0x10`) instructions.

Finding entry points. As mentioned in Section 3.2, *iDEA* has a few steps in the hunting for Type-II *entry-points*. In the first step, we find the return instruction that returns a `method-struct`, and exclude those that return NULL. This is because certain branch returns NULL to handle invalid `selector` argument, which intuitively is used to select a `method-struct` from its array. To this end, *iDEA* integrated off-the-shelf tool Triton [62, 65] to taint the register with the "selector" argument and find tainted return instruction – one that returns `method-struct` but not NULL. In the second step, *iDEA* integrated backward slicing provided by Triton to find instructions that affect the value in the return register (`X0` for iOS/iPadOS/tvOS, `rax` for macOS). In these instructions, we look for the address loading instructions (`ADRP` for iOS/iPadOS/tvOS, `lea` for macOS). If the instruction is loading an address in the data section, we take it as the address of `method-struct` array.

To parse the array, we first need to decide the array length. To this end, we performed symbolic execution (again using Triton) on the `selector` argument along the control flow path leading to the address loading instruction (found above): because the address loading happens after the *user-entry* confirms `selector` does not exceed the array length, the symbolic execution analysis of Triton gives us the range constraints on the `selector`, i.e., the array length. After the length of the `method-struct` array is found, we traverse the array and extract function pointers (i.e., Type-II entry points) in each `method-struct`. For the array found in `get*Target*ForIndex` *user-entries* (see Table 2), each `method-struct`'s size is 48 and the function pointer is placed at offset 8. For `externalMethod` *user-entry*, each `method-struct`'s size is 24 and the function pointer is placed at offset 0; other *user-entries* do not involve `method-struct` and Type-II entry points.

Identifying object types. To decode the mangling name (Scenario 3), which are encoded in Itanium C++ mangling format [30] in Apple drivers, *iDEA* leveraged IDA pro's python API `Demangle()`. Further, in order to get the types of kernel APIs' return values (Scenario 4), we developed a tool based on libclang [75] to parse the header and source files in the XNU source code.[2] This tool collects kernel functions' types (including argument types and return value types) for *iDEA*. Besides, for Scenario 5, the constructor code patterns we used for Apple drivers are listed in Table 4 in Appendix.

Supporting pluggable policy checkers. To support pluggable checkers, *iDEA* provides a template class for checkers to inherit from. Checkers are implemented in Python, the same as *iDEA*. Checkers must override two functions (`pre_instr_checker` and

[2] Apple released some versions of XNU [7]. The types of kernel functions typically remain stable between OS versions for backward compatibility.

post_instr_checker), which are invoked by *iDEA* before and after each instruction when traversing the ICFG. *iDEA* employs two analysis clients (TAC and SEC, see Section 3.5), and their analysis results (tainted register/memory area, symbolic expressions, etc.) are provided to checkers through these callback functions.

10 PERFORMANCE EVALUATION

Overview. We use the same set of drivers in Section 5.2 (870 driver binaries on the latest versions of four Apple OSes) to evaluate the analysis time of *iDEA*. We ran *iDEA* to scan all these drivers on a Macbook Pro with 2.2 GHz Intel Core i7 CPU and 16 GB 1600 MHz DDR3. The analysis time is shown in Table 3, which includes the time for all phases of *iDEA* (Section 3) and running two checkers (Section 4). It takes 6~14 hours to analyze an OS, depending on its number of driver binaries. Note that, given a driver binary, Phase I-IV only needs to run once and the results can be stored by *iDEA* to run more checkers.

Table 3: Analysis time for different Apple OSes (in hours)

OS	Number of Drivers (Binaries)	Size (MB)	Overall Time	Time of Phase I-IV	Time to Run Checkers
macOS	362	525	14	10	4
iOS	176	21	7	5	2
iPadOS	189	23	8.5	6	2.5
tvOS	143	13	6	4	2

Table 3 also shows the total size of driver binaries on each Apple OS. Figure 9 further outlines the distribution of driver binary sizes on each Apple OS. For macOS, except for 30 (out of 362) graphics driver binaries taking up 482 MB (92%) of the total size, most driver binaries have a size of 10~100 KB, which is consistent with iOS, iPadOS and tvOS. The 30 graphics binaries have huge data sections (387 MB in total), which did not obviously affect our analysis time. As discussed below, we observed that the major factor affecting analysis time is the size of code section, not data section.

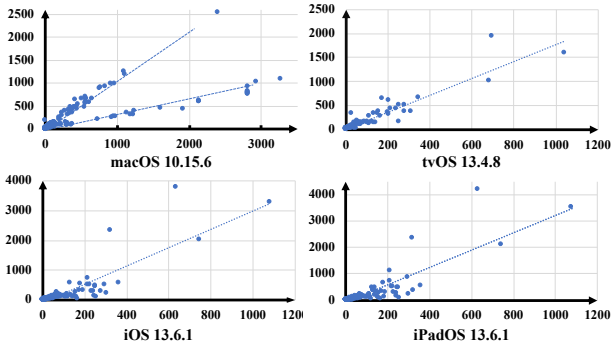

Figure 8: The relationship between code section size and analysis time for macOS, tvOS, iOS, and iPadOS driver binaries, with each dot denoting a driver binary (X-ray: the size of driver binaries' code sections in KB; Y-ray: analysis time in seconds)

Factors affecting analysis time. From a high-level view, the analysis time is roughly proportional to the number of driver binaries, analyzing about 25 driver binaries per hour on each Apple OS (see Table 3). On each OS, we also measured the time for analyzing

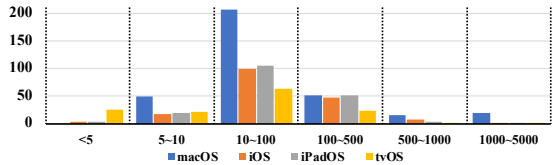

Figure 9: The distribution of driver binary sizes (X-ray: Sizes in KB; Y-ray: Number of driver binaries)

different driver binaries of different sizes and functionalities. The analysis time is mainly affected by the binary size, in particular, the code section of a binary. For most driver binaries, there is an approximately positive linear correlation between the binary size and the analysis time, as shown in Figure 8 . Those sparse dots relatively far above the dotted line have more indirect calls and more complex control flows, and, thus, took more time to analyze. For macOS drivers we observed two linear correlations in Figure **??**. Most drivers around the lower dotted line are related to AMD-manufactured hardware devices (indicated by their binary names such as "AMD9500Controller"). These AMD-related drivers took less time to analyze since they have less indirect calls.

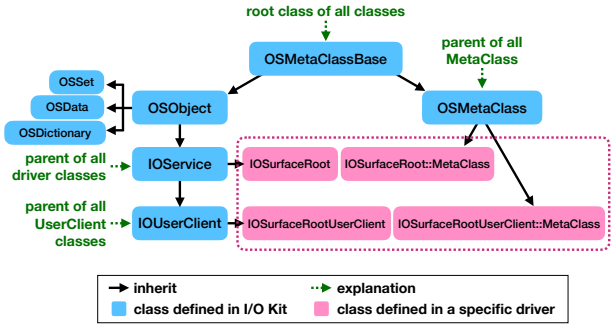

Figure 10: The inheritance relationship between driver classes and classes in the kernel driver programming framework I/O Kit. `OSMetaClassBase` is the ancestor of all classes.

Table 4: Common code patterns of constructors

Function	Size & # of basic blocks	Other features
Constructors (Paradigm 1)	36 & 1	A vtable pointer is loaded at function offset +0xC.
Constructors (Paradigm 2)	72 & 1	`OSMetaClass::instanceConstructed` is called at offset +0x34. A `MetaClass` object is referenced at +0x10

Table 5: Drivers with source code

Source Project Name	Bundle ID
IOPCIFamily	com.apple.iokit.IOPCIFamily
IOStorageFamily	com.apple.iokit.IOStorageFamily
IONetworkingFamily	com.apple.iokit.IONetworkingFamily
IOHIDFamily	com.apple.iokit.IOHIDFamily

Algorithm 1: The algorithm of identifying objects and type propagation

```
1  Funciton
     TypeAnalsisInFunc(f:Function, τ:Map, Δ: Set):
2      if ¬ (f in Δ) then
3          if f is entry_point then
4              τ ⟵ ∅;
5          Γ ⟵ ∅;
6          deFuncName ⟵ demangle function name;
7          f.argTypes ⟵ extractArgTypes(deFuncName);
8          foreach arg_i in f.args do
9              τ [f.arg_i] ⟵ f.argType_i;
10         end
11         fistBB ⟵ f.firstBasicBlock;
12         TypePropagateInBB(τ, fistBB, Γ, Δ);
13     markVisited(f, Δ);
14 end
15 Funciton
     TypePropagateInBB(τ:Map, bb:BasicBlock, Γ: Set, Δ: Set):
16     if ¬ (bb in Γ) then
17         σ ⟵ copy(τ);
18         foreach instruction i in bb.instructions do
19             if i.operator is call then
20                 F ⟵ i.target;
21                 if F is TypeCastFunc then
22                     resolve destClass argument;
23                     σ [x0] ⟵ destClass *;
24                 else if F is allocClassWithName then
25                     resolve className argument;
26                     σ [x0] ⟵ className * ;
27                 else if F is Constructor then
28                     objClass ⟵ classOfConstrucot(F) ;
29                     σ [x0] ⟵ objClass * ;
30                 else if F is KernelAPI then
31                     retType ⟵
                        getRetTypeOfAPI(KernelAPI) ;
32                     σ [x0] ⟵ retType ;
33                 else
34                     TypeAnalsisInFunc(F, σ, Δ);
35             else if i.operator is mov/store/load then
36                 if i.src in σ then
37                     σ [i.dst] ⟵ σ [i.src];
38                 else if i.dst in σ then
39                     σ [i.dst] ⟵ ∅;
40         end
41         markVisited(bb, Γ);
42         while more bb.nextBB do
43             TypePropagateInBB(σ, bb.nextBB, Γ, Δ);
44         end
45 end
```

Table 6: Vulnerabilities found by *iDEA* in Apple drivers ("Silently fixed" means the vulnerability was found in older OS versions, but have been fixed silently by Apple in newer OS versions without public disclosure. "Null-Pointer" means Null-Pointer dereference vulnerability.)

Vulnerable driver	UserClients	OS	Vuln Type	Status
IOFirewireFamily	IOFirewireUserClient	macOS	UAF	CVE-2018-4135
IOMikeyBusFamily	IOMikeyBusDeviceUserClient	iOS/iPadOS	UAF	CVE-2020-3834
IOUSBDeviceFamily	IOUSBDeviceInterfaceUserClient	iOS	UAF	CVE-2019-8836
AppleC26Charger	AppleC26ChargerUserClient	iOS/iPadOS/tvOS	UAF	CVE-2020-3858
IOThunderboltFamily	IOThunderboltFamilyUserClient	macOS	UAF	CVE-2020-3851
AppleFWAudio	AppleFWAudioUserClient	macOS	UAF	Acknowledged by Apple; CVE scheduled to assign
AppleFWAudio	AppleMLANAudioUserClient	macOS	UAF	Acknowledged by Apple; CVE scheduled to assign
AppleIntelSKLGraphics	IGAccelCommandQueue	macOS	OOB read&write	Reported
AppleIntelKBLGraphics	IGAccelCommandQueue	macOS	OOB read&write	Reported
AppleIntelBDWGraphics	IGAccelCommandQueue	macOS	OOB read&write	Reported
AppleIntelHD5000Graphics	IGAccelCommandQueue	macOS	OOB read&write	Reported
AppleIntelHD4000Graphics	IGAccelCommandQueue	macOS	OOB read&write	Reported
AppleMultitouchSPI	AppleMultitouchSPIUserClient	iOS/iPadOS	Null-Pointer	Reported
EncryptedBlockStorage	EncryptedMediaFilterUserClient	iOS/iPadOS/tvOS	Null-Pointer	Reported
LSKDIOKitMSE	com_apple_driver_KeyDelivery IOKitUserClientMSE	iOS/iPadOS/tvOS	Null-Pointer	Reported
LightweightVolumeManager	LwVMUserClient	iOS/iPadOS	Null-Pointer	Reported
AppleSMCLMU	AppleLMUClient	macOS	Null-Pointer	Reported
AppleGFXHDA	AppleGFXHDADriverUserClient	macOS	Null-Pointer	Reported
AppleGFXHDA	AppleGFXHDAControllerUserClient	macOS	Null-Pointer	Reported
AppleImage4	AppleImage4UserClient	macOS	Null-Pointer	Reported
IOBluetoothFamily	IOBluetoothRFCOMM ConnectionUserClient	macOS	Null-Pointer	Reported
IONVMeFamily	AppleNVMeSMARTUserClient	macOS	Null-Pointer	Reported
AppleGraphicsPowerManagement	AGPMClient	macOS	Null-Pointer	Reported
watchdog	IOWatchdogUserClient	macOS	Null-Pointer	Reported
AppleACPIPlatform	AppleACPIPlatformUserClient	macOS	Null-Pointer	Reported
AppleSMBusController	AppleSMBusControllerUserClient	macOS	Null-Pointer	Reported
AppleHDA	AppleHDADriverUserClient	macOS	Null-Pointer	Reported
SMCMotionSensor	SMCMotionSensorClient	macOS	Null-Pointer	Reported
IOSCSIBlockCommandsDevice	AppleNVMeTranslation SMARTUserClient	macOS	Null-Pointer	Reported
ACPI_SMC_PlatformPlugin	ACPI_SMC_PluginUserClient	macOS	Null-Pointer	Reported
IOHDAFamily	IOHDACodecDeviceUserClient	macOS	Null-Pointer	Reported
AppleMikeyDriver	AppleMikeyDriverUserClient	macOS	Null-Pointer	Reported
AppleHDAController	AppleHDAControllerUserClient	macOS	Null-Pointer	Reported
IOATABlockStorage	ATASMARTUserClient	macOS	Null-Pointer	Reported
AppleDiskImagesKernelBacked	KDIDiskImageNubUserClient	macOS	Null-Pointer	Reported
IOUserEthernet	IOUserEthernetResourceUserClient	macOS	UAF	Silently fixed
AppleIntel8254XEthernet	Intel8254XUserClient	macOS	UAF	Silently fixed
mDNSOffloadUserClient	mDNSOffloadUserClient	macOS	UAF	Silently fixed
IOAVBStreamingPlugin	IOAVBInputUserSpaceStreamUserClient	macOS	UAF	Silently fixed
IOUSBDeviceFamily	IOUSBDeviceInterfaceUserClient	iOS	UAF	Silently fixed
AppleJPEGDriver	AppleJPEGDriverUserClient	iOS	Null-Pointer	Silently fixed
IOUserEthernet	IOUserEthernetResourceUserClient	iOS	Null-Pointer	Silently fixed
AppleS7002SPU	AppleSPUHIDDeviceUserClient	iOS	UAF	Silently fixed
EncryptedBlockStorage	EncryptedMediaFilterUserClient	iOS	Null-Pointer	Silently fixed
LSKDIOKit	com_apple_driver_ KeyDeliveryIOKitUserClient	iOS	Null-Pointer	Silently fixed
IOImageLoader	IOImageLoaderUserClient	iOS	Null-Pointer	Silently fixed

Listing 2: The POC of UAF vulnerability in `IOFirewireUserClient`

```
io_connect_t connection = (io_connect_t) 0;
uint64_t outputHandle = 0xaa;

void race(void *args){
  kern_return_t kr;
  struct CommandSubmitParams inputStruct = {0};
  inputStruct.type = kFireWireCommandType_Read;
  size_t outputStructCnt = 10;
  uint64_t asyncRef[8] = {0};
  uint32_t selector;
  struct CommandSubmitResult outputStructNew = {0};
  outputStructCnt = sizeof(outputStructNew);
  inputStruct.kernCommandRef = outputHandle;
  selector = kCommand_Submit;
  kr = IOConnectCallAsyncMethod(connection, selector, MACH_PORT_NULL, asyncRef, 3, NULL, 0, (void *)&inputStruct,
      ↪  sizeof(inputStruct), NULL, 0, &outputStructNew, &outputStructCnt);
}

void main()
{
  io_service_t service = IOServiceGetMatchingService(kIOMasterPortDefault, IOServiceMatching("
      ↪ IOFireWireLocalNode"));
  kern_return_t kr = IOServiceOpen(service, mach_task_self(), 0, &connection);
  while(1){
    uint64_t input[3] = {0};
    struct CommandSubmitParams inputStruct = {0};
    inputStruct.type = kFireWireCommandType_Read;
    uint64_t output[16] = {0xaa};
    uint32_t outputCnt = 2;
    size_t outputStructCnt = 1;
    uint64_t asyncRef[8] = {0};
    uint32_t selector;
    selector = kCommandCreateAsync;
    kr = IOConnectCallAsyncMethod(connection, selector, MACH_PORT_NULL, asyncRef, 3, NULL, 0, (void *)&
        ↪ inputStruct, sizeof(inputStruct), NULL, 0, &outputHandle, &outputStructCnt);
    pthread_t t;
    pthread_create(&t, NULL, (void*) race, NULL);
    struct CommandSubmitResult outputStructNew = {0};
    outputStructCnt = sizeof(outputStructNew);
    inputStruct.staleFlags = 1<<1;
    inputStruct.kernCommandRef = outputHandle;
    selector = kCommand_Submit;
    wait_n(200);
    kr = IOConnectCallAsyncMethod(connection, selector, MACH_PORT_NULL, asyncRef, 3, NULL, 0, (void *)&
        ↪ inputStruct, sizeof(inputStruct), NULL, 0, &outputStructNew, &outputStructCnt );
    pthread_join(t, NULL);
  }
  IOServiceClose(connection);
}
```