APICraft: Fuzz Driver Generation for Closed-source SDK Libraries

Cen Zhang, *Nanyang Technological University;* Xingwei Lin, *Ant Group;* Yuekang Li, *Nanyang Technological University;* Yinxing Xue, *University of Science and Technology of China;* Jundong Xie, *Ant Group;* Hongxu Chen, *Nanyang Technological University;* Xinlei Ying and Jiashui Wang, *Ant Group;* Yang Liu, *Nanyang Technological University*

https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

APICRAFT: Fuzz Driver Generation for Closed-source SDK Libraries

Cen Zhang[§] Xingwei Lin[‡] Yuekang Li [§ *] Yinxing Xue[†] Jundong Xie[‡]
Hongxu Chen [§] Xinlei Ying [‡] Jiashui Wang [‡] Yang Liu [§]
[§]*Nanyang Technological University* [‡]*Ant Group* [†]*University of Science and Technology of China*

Abstract

Fuzz drivers are needed for fuzzing libraries. A fuzz driver is a program which can execute library functions by feeding them with inputs provided by the fuzzer. In practice, fuzz drivers are written by security experts and the drivers' quality depends on the skill of their authors. To relieve manual efforts and ensure test quality, different techniques have been proposed to automatically generate fuzz drivers. However, existing techniques mostly rely on static analysis of source code, leaving the fuzz driver generation for closed-source SDK libraries an open problem. Fuzz driver generation for closed-source libraries is faced with two major challenges: 1) only limited information can be extracted from the library; 2) the semantic relations among API functions are complex yet their correctness needs to be ensured.

To address these challenges, we propose APICRAFT, an automated fuzz driver generation technique. The core strategy of APICRAFT is *collect – combine*. First, APICRAFT leverages both static and dynamic information (headers, binaries, and traces) to collect control and data dependencies for API functions in a practical manner. Then, it uses a multi-objective genetic algorithm to combine the collected dependencies and build high-quality fuzz drivers. We implemented APICRAFT as a fuzz driver generation framework and evaluated it with five attack surfaces from the macOS SDK. In the evaluation, the fuzz drivers generated by APICRAFT demonstrate superior code coverage than the manually written ones, with an improvement of 64% on average. We further carried out a long-term fuzzing campaign with the fuzz drivers generated by APICRAFT. After around eight month's fuzzing, we've so far discovered 142 vulnerabilities with 54 assigned CVEs in macOS SDK, which can affect popular Apple products such as `Safari`, `Messages`, `Preview` and so on.

1 Introduction

Fuzzing has become one of the most popular vulnerability detection techniques for both practitioners and researchers since it was introduced in the 1990s [1, 2]. The state-of-the-art fuzzers, like AFL [3], libFuzzer [4] and Honggfuzz [5], have detected more than 16000 vulnerabilities in hundreds of real-world software programs and libraries [6].

To fuzz a program, the fuzzer needs to find an entry-point to which it can feed inputs. To fuzz a library, the fuzzer needs an application program of the library to serve as the entry-point. This application program is called a *fuzz driver*, aka *fuzz harness*[1]. In practice, the creation of fuzz drivers is primarily a manual effort for security analysts. The quality of a fuzz driver depends on the skill as well as knowledge of its writer. Thus, creating effective fuzz drivers is often a time-consuming and challenging task.

To tackle the problem of fuzz driver generation, researchers have proposed some techniques such as FUDGE, FUZZGEN [7, 8]. These techniques leverage the source code of existing consumer applications of a library to synthesize fuzz drivers for it. On one hand, the source code of consumer programs provides correct API usage examples which is important for fuzz drivers; on the other hand, the need for source code limits the usage of these techniques on the libraries in closed-source SDKs. Nevertheless, the security of the closed-source SDK libraries is equally, if not more, important due to their popularity and market dominance. Taking `Preview` as an example, it can display various types of files, backed by the different file parsing libraries of the macOS SDK. Since `Preview` is pre-installed on every Apple PC/laptop, vulnerabilities found in the related libraries can affect millions of end users. In summary, fuzz driver generation for closed-source SDK libraries is an important yet understudied problem.

The challenges of fuzz driver generation for closed-source SDKs come from two main aspects. ❶ The first challenge is that only limited information can be extracted from the library. The absence of library source code hinders the extraction of correct API usages required for fuzz driver synthesis. Worse still, the consumer programs are often closed-source, for example, `Preview`. Due to some notorious pitfalls such

*Corresponding Author.

[1]Fuzz driver and fuzz harness refer to the same thing. For consistency, we only use the term fuzz driver in this paper.

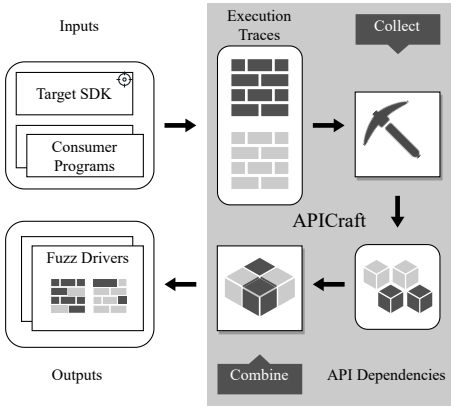


Figure 1: The overview of APICRAFT.

as indirect function calls, the control-flow and data-flow information of the library APIs cannot be extracted accurately without source code. ❷ The second challenge is that the semantic relations among API functions are complex yet their correctness needs to be ensured. Triggering the code deep in a library requires the test drivers to contain semantically correct API call sequences. In fact, not only is the search space for API call combinations is huge, but also the semantic correctness of the API call sequences is hard to guarantee.

To address the aforementioned challenges, we propose APICRAFT [2] by answering the questions of what information to extract and how to utilize it. Fig. 1 illustrates the overall structure of APICRAFT. Basically APICRAFT takes as inputs the target SDK and its consumer programs and produces fuzz drivers as output. APICRAFT uses a bottom-up approach to synthesize fuzz drivers which can be described as a *collect – combine* method. This approach consists of two main stages. The first stage is the collection of the dependencies among API functions in the target SDK libraries. Here, APICRAFT uses the execution traces of the target SDK's consumer programs as the reference of correct API usages. Instead of collecting every data and control dependency, which is impractical, APICRAFT only collects the inter-API-function data dependency and control dependency related to error handling. The second stage is to combine the collected API function dependencies to build the fuzz driver suite with desired properties, such as compactness and dependency diversity. Since the desired properties can conflict with each other, APICRAFT uses a multi-objective genetic algorithm based strategy to optimize the whole population of fuzz drivers towards satisfying a predefined set of objectives.

We implemented APICRAFT as a framework for generating fuzz driver suites for closed-source SDKs and conducted thorough evaluations. We evaluated the fuzz drivers generated by APICRAFT with five attack surfaces from macOS SDK.

[2] APICraft, ['eɪpɪkra:ft], stands for API plus Minecraft (a sandbox video game in which the players can craft cool stuff with basic building blocks).

In our experiments, we found that the generated fuzz drivers outperform their manually written counterparts in both code coverage (64% more basic blocks on average in 24h) and unique crashes (12 more unique crashes on average in 24h). Moreover, we conducted a long-term fuzzing campaign with the generated fuzz drivers. By far, 142 vulnerabilities have been detected (54 CVEs) in macOS SDK which affect some well-known COTS products such as `Safari`, `Preview`, etc.

Contribution. In summary, our contributions are:

- We identified the key challenges of fuzz driver generation for closed-source SDKs and proposed the *collect – combine* approach.
- We developed APICRAFT as the first automatic fuzz driver generation framework for closed-source SDKs, which shows capabilities for testing real-world applications.
- We evaluated APICRAFT on macOS SDK and discovered 142 previously unknown vulnerabilities. We responsibly disclosed them and helped the vendor to fix them.

To facilitate future research, we release APICRAFT's source code and the generated fuzz drivers in [9].

2 Roadmap

2.1 A Practical Example

Consider the following scenario: Jane is a security analyst and she is given a task of fuzzing a close-sourced library from the macOS SDK, say the `CoreText` library, what would she do to carry out the task? Jane first needs to figure out the functionality of the library. In this case, `CoreText` is a font rendering library. Then, she will try to find a program which uses the library to see if the program is eligible as a fuzzing driver. She may find that `Messages` and `Safari` are using the library. Unfortunately, these apps are also closed-source and involve heavy GUI interactions during execution, meaning that they are not suitable to serve as fuzz drivers. *As a result, Jane has to create custom fuzz drivers for* `CoreText`.

Let's assume, instead of trying to learn from the documentation, Jane would like to create the fuzz drivers by learning how existing consumer programs use the library functions. Because almost all consumer programs of the `CoreText` library are closed-source complex commercial software, *it is hard to extract the correct usage of library functions through disassembling and static analysis.* Alternatively, Jane can use the execution traces of the consumer programs to infer the correct sequences of calling the library functions. In this sense, Jane can build a tiny consumer program based on each execution trace. Assume Jane has gotten two consumer programs as shown in Fig. 2. In `CoreText`, `Font` is an opaque type holding the parsed font data. It can be extracted from either a `FontDescriptor` or a `DataProvider` which can be created with raw font data (`data`). With the `Font` object, `CoreText` can perform many operations, for example, calculating the

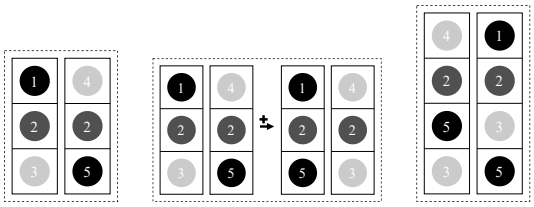
```
1 DataProvider* prov = ProviderCreateWithData(data);  ❶
2 Font* font = ExtractFont(prov);  ❷
3 DoubleLeadingSpace(font);  ❸
```

(a) Consumer 1

```
1 FontDescriptor* desc = CreateFontDescriptor(data);  ❹
2 Font* font = ExtractFont(desc);  ❷
3 CalcLeadingSpace(font);  ❺
```

(b) Consumer 2

Figure 2: The consumer programs of `CoreText`. Some Details are omitted here for the concern of conciseness.



(a) Carved (b) Crossover (c) Desired

Figure 3: Potential fuzz driver suites (rectangles with dashed border) generated based on the consumer programs. Each circle represents a function in Fig. 2. Each chain of circles represents a fuzz driver (made up of function calls).

leading space of the glyph (`CalcLeadingSpace`) or doubling the leading space (`DoubleLeadingSpace`).

Once Jane has collected the knowledge about the correct usage of `CoreText`, the next step is to use the knowledge to build fuzz drivers. Naturally, the two consumer programs carved from execution traces can be used as fuzz drivers, as shown in Fig. 3a. However, Jane will soon notice that the directly carved fuzz drivers are not ideal due to the lack of diversities for the covered program behaviors. In this example, the function `ExtractFont` serves as a pivot connecting the creation of the `Font` stub and the usage of it. Specifically, both `CalcLeadingSpace` and `DoubleLeadingSpace` take a `Font` object as input. As a result, Jane can swap these two functions to create fuzz drivers with new combinations of functions. Fig. 3b shows the fuzz drivers generated by swapping the functions according to the pivot point. Despite containing more combinations of the API functions than the carved fuzz drivers, the fuzz drivers generated with crossover are still far from ideal: ❶ Some API function combinations are still missing. For example, both `DoubleLeadingSpace` and `CalcLeadingSpace` are using the result of `ExtractFont`. Instead of replacing each other, they can be put together into one fuzz driver and trigger more program behaviors. In this case, if `DoubleLeadingSpace` is executed first, `CalcLeadingSpace` may run into an Integer-Overflow bug. ❷ Some combinations are redundant. For example, the two new combinations introduced in Fig. 3b do not really trigger more program behaviors. The reason is how the program uses the `Font` object is normally not affected by how it is generated and calling either `CreateFontDescriptor` or `ProviderCreateWithData` will end up in generating the same `Font` object. In short, the fuzz driver suite built by crossover lacks both diversity and compactness. However, in most cases, these two desired properties are independent and may conflict with each other. *Therefore, building a set of desired fuzz drivers requires balancing different objectives (e.g., compactness and diversity).*

Now, Jane has realized that the fuzz drivers explicitly extracted from the execution traces need improvements and there are some pitfalls for improving the quality of fuzz drivers. Can she just *break down* all the carved fuzz drivers into dependencies between functions and then *combine* these dependencies to rebuild new fuzz drivers which can fulfill multiple independent objectives? After some reasoning and trial-and-error, Jane will eventually realize that the desired fuzz drivers generated with the consumer programs should be as shown in Fig. 3c because these fuzz drivers are compact yet can trigger the most diverse program behaviors. This marks the end of the whole story.

In fact, in APICRAFT, we systematically depict the entire reasoning as well as trial-and-error process of this story as algorithms and can automatically generate the desired fuzz drivers shown in Fig. 3c.

2.2 Overview

We propose APICRAFT to automatically generate fuzz driver suites for commercial SDK libraries. As shown in Fig. 4, the workflow of APICRAFT contains three main stages: ❶ In pre-processing stage, APICRAFT extracts and groups target SDK's information from multiple sources via several kinds of analyses (i.e., header analysis, static binary analysis, and dynamic binary analysis). It outputs a set of library metadata and the execution traces of consumer programs. ❷ APICRAFT collects data dependencies and control dependencies from the outputs of pre-processing. For data dependencies, APICRAFT focuses on the inter-API-function data dependencies. For control dependencies, APICRAFT recognizes and collects function outputs used for error handling. ❸ APICRAFT applies a multi-objective genetic algorithm to combine the collected dependencies into fuzz drivers and drive the generated fuzz drivers towards the desired properties.

3 Methodology

3.1 API Function Dependency Collection

3.1.1 Data Dependency

Key Concepts APICRAFT focuses on the inter-function data dependencies. For a given API function F, we denote

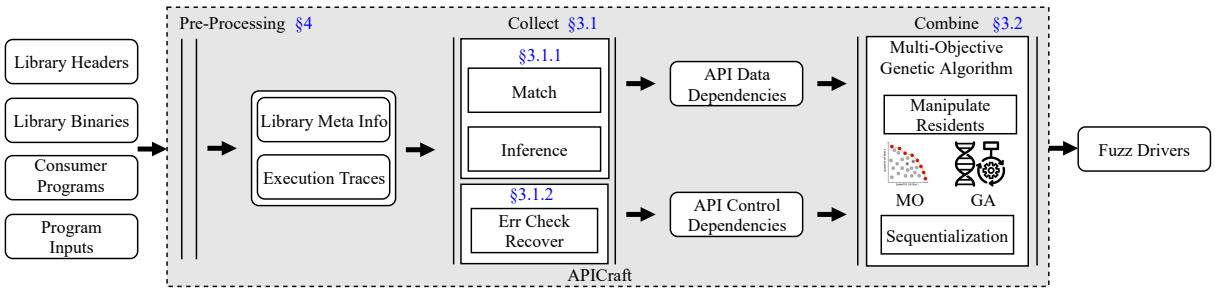


Figure 4: The workflow of APICRAFT.

its input set as I_F, and its output set as O_F. These two sets represent the data F consumes and produces respectively. Given two functions F_A and F_B, they have data dependency if and only if $(I_{F_A} \cap O_{F_B}) \cup (I_{F_B} \cap O_{F_A}) \neq \emptyset$. Specifically, if F_B depends on F_A, the data dependency will be denoted as the tuple $\langle F_A, Out, F_B, In \rangle$ where $Out \in O_{F_A}$ and $In \in I_{F_B}$.

In general, data dependencies can have numerous forms. For instance, function A can depend on function B by reading a socket that B writes to. Different forms of data dependencies require different detection and collection techniques. Currently, APICRAFT uses the following two kinds of data dependencies.

- The return value of function A is used as an input parameter of function B.
- The output parameter (normally in the form of a pointer) [10] of function A is used as an input parameter of function B.

In APICRAFT, for a function F, I_F is the input parameter set, O_F includes its return value and output parameters (if any).

Extraction The aforementioned data dependencies can be extracted by matching the type and value of the API function parameters/return values. We first discuss the collection of type and value information, then detail the extraction process.

The type information is collected from the SDK's header files. By analyzing the function declarations inside the headers, APICRAFT collects the type of the parameters and return value for each API function. Then, it traces the consumer programs to get value information. By hooking the API function's entry and exits during the execution, APICRAFT records thread id, nested level, and the recursive memory dump of its input and output set (i.e., parameters, return value, and output parameters). Here, the term nested level is used to represent the depth of nested API function calls. If an API function is called directly from consumer programs rather than some API functions, its nested level is 1. If an API function is called from another API function with a nested level of x, then its nested level is $x+1$. The recursive memory dump of a parameter or return value is obtained by ❶ dumping its value directly if it is neither a pointer nor a `struct`; ❷ dumping each member's value if it is a `struct`; ❸ dumping its value and its pointee's value if it is a non-null pointer.

Algorithm 1 Basic Data Dependency Extraction

Input: T (An API function trace)
Output: R (Data dependency set)

```
1:  R ← ∅
2:  cache ← { }
3:  for F_B ∈ T
4:      for In ∈ I_{F_B}
5:          for ⟨F_A, Out⟩ ∈ cache[In.value]
6:              if Out.type ≝^{type} In.type
7:                  R ←+ ⟨F_A, Out, F_B, In⟩
8:      for Out ∈ O_{F_B}
9:          if Out.value ≠ 0
10:             cache ←+ {Out.value : ⟨F_B, Out⟩ }
```

APICRAFT processes the collected traces to efficiently and accurately extract the data dependencies. Note that multiple traces of several consumer programs are collected and each trace contains a list of API functions in execution order. APICRAFT first divides each trace into shorter pieces according to thread id, then filters out the functions whose nested level is not 1. The functions with higher nested levels are considered less important and removed since they are not directly called from the consumer programs. After filtering, APICRAFT identifies the possible output parameters for each API function. Specifically, an input parameter will be marked as an output parameter if it is a pointer and the content it points to changes during the execution of the function.

Algorithm 1 shows the simplified data dependency extraction process. Input T is a piece of processed trace containing a list of executed API functions, and output R is a set of extracted data dependencies. The key idea of this algorithm is that, for any two API functions in the trace (F_A, F_B), APICRAFT tries to find the matched pair $\langle F_A, Out \rangle$ and $\langle F_B, In \rangle$. The pair is matched if and only if F_A's Out has the exact value as F_B's In while the Out's type is equivalent as In's type. APICRAFT skips the matched case whose compared value is zero because mostly the match of empty values is not strong enough to indicate a data dependency. For type comparison, APICRAFT first removes the effect of `typedef`

by comparing the canonical types of the two types in pairs. If they are not the same, it further checks whether these two types are convertible. Same types with different attribute qualifiers (e.g., `const` qualifier) are convertible. Besides, pointer types are also convertible if their pointees' type sizes are equal or one of the them has `void *` type.

Inference Besides extracting dependencies from the execution traces, APICRAFT further infers new dependencies based on existing ones. The basic idea is that the API functions from one SDK usually share the same design or implementation pattern. Therefore, following proper heuristics, new valid data dependencies which do not appeared in consumer program traces can be inferred based on the extracted dependencies. APICRAFT uses the following three inference rules:

- **R1: Dependency-based transition**. Suppose we have observed dependencies $\langle F_A, Out_A, F_C, In_C \rangle$ and $\langle F_B, Out_B, F_C, In_C \rangle$. Then if we meet $\langle F_A, Out_A, F_D, In_D \rangle$, we can generate a data dependency $\langle F_B, Out_B, F_D, In_D \rangle$.
- **R2: Type-based transition**. If we observe $\langle F_A, Out \rangle$, $Out.type \overset{type}{=} T$ and $\langle F_B, In \rangle, In.type \overset{type}{=} T$, we can generate a data dependency $\langle F_A, Out, F_B, In \rangle$.
- **R3: Inter-thread data flow dependency**. By adjusting Algorithm 1 (see Appendix A), we identify inter-thread data flow dependency $\langle F_{A,T1}, Out, F_{B,T2}, In \rangle$ where $F_{A,T1}, F_{B,T2}$ means two functions (A, B) from two threads' ($T1$, $T2$) traces. We only extract inter-thread dependencies where either *In* or *Out* is of pointer type.

In practice, traces cannot contain a full list of data dependencies for its covered API functions. **R1**, **R2** mitigate this limitation. Assuming there are two sets of API functions which create and use a specific type of object respectively and the trace only contains one or two related dependencies, **R1**, **R2** can help to infer all links between these two sets of functions. Another observation is that threads exchange limited data with other threads and usually exchange pointers (e.g, one thread only creates objects for other threads). Therefore, APICRAFT uses **R3** to match pointers between traces from different threads to dig these dependencies out. During the inference, **R3** is firstly applied, then **R1**, **R2** are repeatedly applied until no new data dependency can be generated.

3.1.2 Control Dependency

Besides data dependencies, APICRAFT collects control dependencies to facilitate fuzz driver synthesis. Specifically, APICRAFT collects error handling information. By combining static and dynamic analysis, two types of information are collected: whether an API function's output needs to do error handling and the error condition.

APICRAFT uses different strategies for different types of function's output. If the canonical type of an output parameter or return value of an API function is pointer, the generated fuzz driver will always check if the output value is `NULL` or not (exit immediately if it is `NULL`). If the type is integer, APICRAFT will try to locate the conditional branches where the consumer program applies the error check, and dump the error check conditions (assuming the consumer program will execute the no-error branch under benign input). Firstly, APICRAFT locates the callsite address by recording the API function's return address [3] during tracing. Then, with static analysis, APICRAFT finds the dominator basic blocks (namely checkpoints) of the places where the termination functions (like `_exit`, `_abort`, `__cxa_throw`) are called. Finally, APICRAFT reruns the consumer program, conducts dynamic taint analysis starting from the callsite marking the integer output as taint source. The taint propagation stops when the callsite's function returns or the checkpoint is tainted. The tainted checkpoint will be treated as the error checking branch and its condition will be dumped as the error handling condition. For other types of output, their values will not be checked in the generated fuzz driver.

3.2 Dependency Combination

After collecting the basic dependencies, the next step of APICRAFT is to utilize them to synthesize fuzz drivers. Naturally, data dependencies can be used as building blocks for creating a complex data flow. APICRAFT uses a search-based algorithm to randomly and repeatedly link these data dependencies with proper guidance. Starting from a specified point (input related function), the data flow of a fuzz driver can form a tree. APICRAFT first tries to build trees with better qualities according to several metrics. Then, when APICRAFT sequentializes a tree to the corresponding code sequence (fuzz driver), it uses the control dependencies to improve the robustness of the generated fuzz driver.

3.2.1 Problem Modeling

Identified Key Metrics We identify the following three metrics for measuring the quality of a fuzz driver.

- **M1: Diversity** To sufficiently test the target, the fuzz driver needs to include as many distinct API functions as possible. Besides, the more data dependencies a fuzz driver contains, the more inter-function data exchanges the driver's execution covers, which increases the possibility for uncovering bugs related with erroneous data management in the target SDK during fuzzing.
- **M2: Effectiveness** Besides using more API functions, fuzz driver needs to call these functions correctly. Valid usage of API functions in the driver is a necessity for fuzzing since the generated fuzz driver can report lots of false-positives. Moreover, the correct usage of an API function can help to test more core logic of that function.
- **M3: Compactness** Given two fuzz driver candidates with similar diversity and effectiveness, we prefer the

[3] The return address is extracted using `gcc` built-in function `__builtin_return_address`.

more compact one. A fuzz driver is more compact if it has less duplicate or irrelevant function calls/data dependencies. A more compact fuzz driver is easier to use, understand and debug with, which saves not only manual efforts during analysis but also the computation resources during fuzzing.

All the above features are independent metrics for measuring the fuzz driver, and our aim is to generate a fuzz driver which can perform well for all of them. For this purpose, we can design fitness functions (score formulas) to describe these metrics and apply the genetic algorithm to search for better dependency combinations which have higher scores. The key challenge here is the balance of multiple metrics. If we simply use a single formula (say $k_1 * M_1 + k_2 * M_2 + k_3 * M_3$) to glue these three metrics, we can hardly determine the optimal values for the coefficients(i.e., k_1, k_2, k_3). On one hand, since the metrics are independent with each other, the units of these three metrics' scores are hard to be aligned with each other. On the other hand, the optimal value of the coefficients can vary among different target SDKs or traces. Therefore, to balance these important yet conflicting metrics, we propose a multi-objective optimization (MOO) solution. Specifically, we model the dependency combination problem into a multi-objective genetic algorithm called NSGA-II [11].

3.2.2 Multi-Objective Genetic Algorithm

Notation In APICRAFT, a gene stands for one data dependency, and a chromosome is a set of linked data dependencies (genes). Given a data dependency $\langle F_A, Out, F_B, In \rangle$, if we denote functions F_A, F_B as nodes, a gene is an unidirectional edge from node F_A to F_B. Note that there can exist more than one edge between two nodes as one function can generate multiple outputs (return value and output parameters) and each output can be repeatedly used as any other function's input (if can be used to). Therefore, a chromosome is a directed multigraph [12]. For the algorithm's simplicity, the chromosome whose multigraph is cyclic is abandoned.

NSGA-II The genetic algorithm of APICRAFT (Algorithm 2) is based on NSGA-II [11] which has the same basic workflow as the classic genetic algorithm [13] (line 25-30) except that the chromosome ranking strategy handles multiple objectives (❖ line 11-16). In NSGA-II, an objective is a metric which has a score formula to measure a chromosome from an independent dimension. And each chromosome has more than one objective (aka multi-objective). The basic idea of NSGA II's rank strategy is to select the elite chromosomes in two stages. Assuming there are three objectives, the objective scores can be used to build a three-dimensional coordinate system and chromosomes are points. In the coordinate system, a chromosome is in the outermost layer means there is no chromosome can have higher score than it in all objectives. The first stage rank is to divide the chromosomes into several layers (aka Pareto frontiers [14]) by repeatedly choosing all

Algorithm 2 APICRAFT's Multi-objective genetic algorithm

```
Input: D ( Data dependency set )
Output: F ( Fuzz driver candidate list )
1:  procedure OBJECTIVES-SCORE-CALC(R)                          ▷ ❷
2:      for r ∈ R
3:          c ← sequentialization(r)                            ▷ ❸
4:          if pass-stability-test(c)                           ▷ ❹
5:              r.objs[0] ← objective-EFF-calc(c)
6:              r.objs[1] ← objective-DIV-calc(r)
7:              r.objs[2] ← objective-COMP-calc(r)
8:          else
9:              abandon-resident(r)
10: end procedure
11: procedure RESIDENTS-SELECTION(R)                            ▷ ❖
12:     objectives-score-calc(R)
13:     pareto-frontiers-calc-n-sort(R)
14:     crowding-distance-calc-n-sort(R)
15:     R ← residents-filter-by-max-popu(R)
16: end procedure
17: procedure MAKE-NEW-POPULATION(R)                            ▷ ❺
18:     while not exceed max new population number
19:         p1, p2 ← select-parents(R)
20:         c1, c2 ← crossover(p1, p2)
21:         mutate(c1)
22:         mutate(c2)
23:         R ←+ c1, c2
24: end procedure
25: R ← generate-initial-residents(D)                           ▷ ❶
26: residents-selection(R)
27: while not exceed max round
28:     make-new-population(R)
29:     residents-selection(R)
30: F ← R.pareto-frontiers[0]
```

chromosomes in the outermost layer. The second stage rank is intralayer. A less crowded chromosome will have higher score (calculating the distance of a chromosome with its neighbours in the coordinate system). After rank, top-resident-number of the chromosomes are selected to attend next round's evolution. Finally, the results are all chromosomes on the first Pareto frontier in the final round of the evolution.

❶ **Initial Residents** Before combination, APICRAFT needs to build some minimal fuzz drivers which contain at least one input related API functions. For input related API function, we mean the function which either handles the input file descriptor or directly operates on its content (e.g., `CTFontCreate`). APICRAFT firstly identifies the input related API functions in the target SDK, then tries to build a minimal fuzz driver based on them (fills in all parameters of these functions). The input related functions can be located by matching the key features of the input file, such as matching the dumped API parameter value with input file's name or content. Once

successfully matched, APICRAFT will mark that function and parameter, and pass the input to the corresponding parameter when generating fuzz driver's code. To build the minimal fuzz driver, APICRAFT also needs to fill in the values of other parameters in the input related functions. It searches from three sources to feed the value: output value of another API function, preconfigured basic knowledge (more detail in ❸), or the dumped parameter value. APICRAFT randomly chooses value from the above sources and generates the driver's code. The generated code is compiled and executed with several prepared input seeds (we call this stability test, detailed in ❹). Once the test is passed, it is a valid minimal fuzz driver. After building one or more minimal fuzz drivers, APICRAFT iterates all data dependencies, tries to link them with the drivers, and sets the linked ones as initial residents.

❷ Objectives We design three score formulas to describe the identified three metrics in Section 3.2.1. We first introduce the concept of core dependency. In a fuzz driver, a dependency $\langle F_A, Out, F_B, In\rangle$ is a core dependency if one of F_A's input is either input data or the output of another core dependency. The core dependencies in the driver are expected to form a top-down tree-like graph representing the input data flow. In other words, the data flow starts from a root node which is an input related API function and the core dependencies help the input data flows into different API functions. All non-core dependencies are used for filling the inputs of functions inside the core dependencies. When calculating the objective scores, we mainly use core dependencies rather than all data dependencies inside a fuzz driver. The rationale of discriminating non-core dependencies which cannot be influenced by the input data is that they are valueless during fuzzing when different inputs are fed into the fuzz driver. Besides, we denote the functions related to core dependencies as core functions.

Diversity metric (*DIV*) is measured using Equation 1. APICRAFT builds a core dependency graph of the fuzz driver to calculate *DIV*. The score is composed of two parts: E and CC. E is the number of distinct edges in a graph which measures how many unique core dependencies are used in a fuzz driver. CC is the cyclomatic complexity [15] of the graph which stands for the number of the loops in the graph (more loops means higher complexity). Therefore, *DIV* favors fuzz drivers using more unique and complex data dependencies.

$$DIV = E + CC \tag{1}$$

Effectiveness metric (EFF) is measured using Equation 2. B stands for the covered basic block set. EFF is a metric evaluating the fuzz driver's dynamic behaviour. Intrinsically, it is a weighted basic block coverage. EFF evaluates whether an API function is used correctly or not by giving bonus scores for basic blocks which are in loop or contain function calls. The intuition is that error handling path inside a function contains less basic blocks than the core logic code, since core logic code is more complex, i.e., has more loops or calls.

$$S_{eff}(b) = \begin{cases} 3 & \text{if basic block } b \text{ has call and in loop} \\ 2 & \text{if basic block } b \text{ either has call or in loop} \\ 1 & \text{otherwise} \end{cases}$$

$$EFF = \sum_{b \in B} S_{eff}(b) \tag{2}$$

The compactness metric (*COMP*) is measured using Equation 3. F, f, I_f, i, F_{num} stand for the core function set, a core function, input parameters of f, an input parameter, and the total number of core functions. *COMP* describes compactness from two aspects: less duplicate and less irrelevant usage of data dependencies. This means that *COMP* favors a fuzz driver which contains less non-core dependencies (they are irrelevant as the input data cannot influence them) and avoids redundant use of core dependencies. *COMP* measures the overall compactness of a fuzz driver by measuring the average compactness of all input parameters of core functions inside the fuzz driver's input data flow tree. And it evaluates the input parameter's compactness by evaluating the compactness of its value's source. The source of an input parameter can be from: ① an output of a core function or from, ② preconfigured basic knowledge or memory dump or from, ③ an output of a non-core function. For source ①, it is the most compact case (is part of the core dependency) and has the score 2. For source ②, it is compact (avoids use of non-core function) and has the score 1. For source ③, its compactness depends on how many non-core functions are used to provide this value (a non-core function may require several other functions to feed its input, the total amount of non-core functions is marked as k in Equation 3, and we empirically set its score as 0 when $k >= 5$). For duplicate dependencies, we count its score once. The right part of the numerator in Equation 3 is to normalize *COMP*: if a fuzz driver has no duplicate core dependency, doesn't use any non-core function, and has no circle in its core dependency graph ($CC = 1$), its *COMP* is 1.

$$S_{comp}(i) = \begin{cases} 2 & \text{if } i \text{ is in core dependency} \\ 1 - \frac{min(k,5)}{5} & \text{if } i \text{ is in non-core dependency} \\ 0 & \text{if } S_{comp}(i) \text{ has been counted} \\ 1 & \text{otherwise} \end{cases}$$

$$COMP = \frac{\sum_{f \in F} \sum_{i \in I_f} S_{comp}(i) - (F_{num} - 1)}{\sum_{f \in F} \sum_{i \in I_f} 1} \tag{3}$$

❸ Sequentialization During the evolution, a fuzz driver is in the form of multigraph based on which APICRAFT applies the mutation operations. Sequentialization is to convert the graph into code (sequences of API function calls). The fuzz driver code is then used in dynamic information collection (for calculating EFF) and validity testing (by compilation and execution) during the evolution. Sometimes, certain data

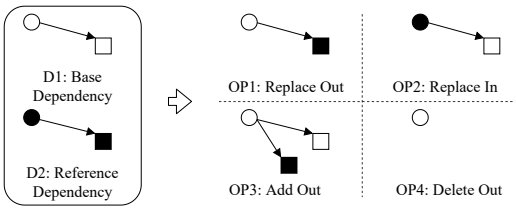


Figure 5: Possible mutation operations (OP1-4) for a given data dependency (D1), different $\langle Func, In/Out \rangle$ are simplified to circles and rectangles with different colors.

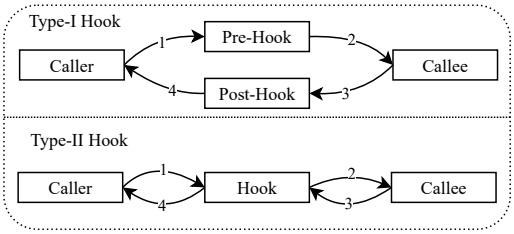


Figure 6: Two types of binary function hook mechanism.

dependencies cannot be collected by APICRAFT, e.g., they are language-builtin knowledge or from other libraries which cannot be acquired by just analyzing the target SDK. This can lead to the compilation error for the sequentialized code as some functions have incomplete inputs. To mitigate this, we provide a manually built basic knowledge and use a lazy update strategy to maintain it in APICRAFT. From our experience, the required amount of basic knowledge is small (more detail in Section 6.1, Appendix D).

❹ **Stability Test** After sequentialization, APICRAFT applies stability test to the fuzz drivers. Stability test is to run the compiled fuzz driver several times using multiple input seeds. The fuzz driver is tested with sanitizers, e.g., ASAN [16] and `libgmalloc` [17]. Usually, APICRAFT uses 3-5 distinct input seeds for a stability test. Once the test fails, i.e., the driver crashes or exits abnormally, it will be abandoned. This test improves the quality of fuzz drivers by filtering unstable data dependencies during the evolution.

❺ **Crossover & Mutate** Fig. 5 shows all mutation operations between two data dependencies. A mutation operation is a combination of Replace/Add/Delete action with In/Out of a data dependency. Note that the combination of *Add/Delete In* is excluded since neither passing more than one value to nor removing the value of an input parameter is a meaningful operation. Besides, the mutation operations in Fig. 5 are simplified as they only consider two given dependencies. Applying these operations to two fuzz drivers requires properly handling other dependencies in the drivers. For instance, both the circle/rectangle nodes can have other data dependencies (have parent/child nodes). APICRAFT guarantees the operation is correctly conducted by carefully handling these cases.

We define crossover and mutate based on the above operations. Crossover is an operation which exchanges genes (data dependencies) between two chromosomes (fuzz drivers). In APICRAFT, crossover is the process of applying one operation (*Replace In*, *Replace Out*, and *Add Out* except *Delete Out*) to two parent chromosomes. It randomly selects an operation and two applicable genes (D1 and D2 from two parents), then applies that operation to both parents to generate two new children chromosomes. D1 and D2 are applicable genes when their input/output satisfies the condition for applying a specific operation. For example, assuming D1 is $\langle F_A, Out, F_B, In \rangle$, D2 is $\langle F_C, Out, F_D, In \rangle$, to apply Replace Out, $\langle F_D, In \rangle$ has to be the same as $\langle F_B, In \rangle$. Mutation is an operation which changes part of a chromosome's genes. To conduct mutation, APICRAFT first randomly selects an operation (all four OPs in Fig. 5 are applicable) and a gene (D1) from the chromosome as the mutate target, and finds an applicable gene (D2, no need if the operation is Delete Out) from the global gene list (the collected dependency set). Next APICRAFT randomly builds a temporary chromosome by satisfying all input parameters of D2, and applies the operation to the original chromosome (similar as crossover with the temporary chromosome but only one child is kept).

4 Implementation

APICRAFT is implemented as a system with three main components, the pre-processing (1,581 lines of Python, 873 lines of C++, 450 lines of Bash), the dependency collection (716 lines of Python, 182 lines of Bash), and the dependency combination (3,749 lines of Python, 93 lines of Bash). Instead of discussing every detail about APICRAFT's implementation, we only discuss some interesting technical details here.

Consumer Program Tracing Tool APICRAFT uses a customized API tracing tool for tracing a series of information (e.g., thread id, nested level, memory dump) during preprocessing. This tracing tool is capable of handling GUI programs in macOS, including `Safari`, `Preview`, `QuickTime Player`, etc. It can generate thousands lines of code to hook hundreds of functions while ensuring the GUI programs to execute smoothly during the tracing. Compared with existing dynamic hook/instrumentation tools (e.g., Pin [18], Frida [19]), it is faster and more accurate. The following key features can improve its performance. ❶ We choose the Type-II hook (Fig. 6) which provides an accurate function nested level. When hooking a function, the Type-I hook requires two kinds of hook points which are one enter point and the exit points. In binary analysis, identifying the start (enter point) of an API function is easy while accurately identifying all exit points is hard. The reason is some exit points of a function cannot be detected by simply matching the `ret` instruction, especially when its assembly is highly optimized. Once the traced program returns from a missed exit point, the following recorded nested level will be corrupted. In contrast, using Type-II hook

doesn't need to concern about this issue since the function returns to the hook code. ❷ We use a lightweight hook technique called function interposition. The hook is accomplished by wrapping hook code into a function which has the same prototype as the hook target and setting environment variables to configure the OS's dynamic linker. Specifically, in macOS, we set `DYLD_PRELOAD`, `DYLD_INTERPOSE` for the hook[4].

5 Evaluation

Our evaluation targets on answering the following questions:

- Can APICRAFT generate fuzz drivers for complex commercial SDK targets (Section 5.1)?
- What is the fuzzing performance of the generated fuzz drivers (Section 5.2)?
- How does each component contribute to the generated fuzz driver's performance (Section 5.3)?
- Can the generated fuzz drivers help to find new vulnerabilities from real-world applications (Section 5.4)?

Hardware Configuration The experiments are conducted on a macOS server with a 2.5GHz 28-core Intel Xeon W processor and 192GB memory.

Attack Surfaces We use five attack surfaces in macOS SDK as the targets, which are `Image`, `Font`, `PDF`, `Audio`, and `RTF`. These attack surfaces accept popular formats of input and have been broadly used by macOS applications. Note that the attack surface and the library are in a many-to-many relationship (see exact mapping is in Appendix B).

Fuzzer Setup In the fuzzing experiments, we use patched `honggfuzz` [5, 20], which can collect the basic block coverage of the target binary libraries. We maintain a seed corpus collected from open Internet resources [21–29] and randomly select seeds from it for the experiments. We minimize this corpus using the `honggfuzz` built-in corpus minimization feature [30]. Since we are fuzzing binary targets in macOS without source code, we can not make use of AddressSanitizer [16] to detect memory error at runtime. Instead, `libgmalloc` [17] is used to detect memory corruption issues when fuzzing.

Fuzzing Experiment Setting Our fuzzing experiment settings are aligned with the suggestions from [31]. For each attack surface, the used fuzz drivers (generated or manually written) share the same input seeds, machine, and fuzzer options (each fuzz driver uses a single thread `honggfuzz` fuzzer with its default options). The plots are drawn using 24 hours, 10 times repeated fuzzing data (lines are average values and shadows along the lines are 95% confidence intervals). As APICRAFT may generate more than one fuzz driver candidate (all fuzz drivers of the first Pareto frontier), we manually select one of them for experiments. Generated fuzz driver selection in Section 5.2, 5.3 follows an empirical selection criteria: choose the one which has better scores in more objectives. And if no one can be better in all three objectives, we follow the priority order DIV > EFF > COMP.

[4]The hook is POSIX-compatible, for linux, `LD_PRELOAD` can be used.

	Image	Font	PDF	Audio	RTF
qlmanage	✓	✓	✓	-	✓
Preview	✓	-	✓	-	-
Font Book	-	✓	-	-	-
Messages	✓	✓	-	✓	-
Safari	✓	✓	-	-	-
Mail	✓	✓	✓	-	-
TextEdit	✓	✓	✓	-	✓
Notes	✓	✓	✓	✓	✓
VoiceMemos	-	-	-	✓	-
Photos	✓	-	-	-	-
Terminal	-	✓	-	-	-
QuickTime Player	-	-	-	✓	-
afclip	-	-	-	✓	-

Table 1: Traced applications when generating fuzz drivers to the attack surfaces. The first row is the attack surface, and the first column is the traced GUI application. ✓ labels a GUI application which can provide trace for an attack surface. - labels a GUI application which doesn't support the input format of an attack surface.

5.1 Fuzz Driver Generation

We applied APICRAFT to five attack surfaces in macOS SDK. Table 2 shows the intermediate results of each major step. The first stage is pre-processing. We select a range of the GUI programs as the consumer programs. Table 1 lists the traced applications for each attack surface. Note that all these programs are built-in macOS applications. We prepared one input file for each GUI program and manually used these programs to generate traces. To generate better traces (containing more diverse dependencies), we guarantee the manual usage of consumer programs covers their all basic features. For example, given an audio player, we at least try to start, pause, forward, backward, randomly jump in an audio's play. Each consumer program is traced using one input file. In theory, using more input files helps in generating a more diverse trace and possibly leads to a better combination result. However, this linearly increases cost in collect stage. To balance between efficiency and effectiveness, we suggest the strategy of tracing the consumer program using one or more representative seeds while exploring diverse consumer program features. The second to fifth columns in Table 2 show the trace information. The fifth column lists the total running time. The majority of the time cost is the trace. Tracing's time cost highly depends on the number of traced applications and APIs. Other pre-processing steps takes small part of the time and can end in minutes. For example, the header analysis usually ends in tens of seconds. APICRAFT *is able to trace hundreds of target API functions in the complicate GUI programs.*

After generating traces, APICRAFT extracts and infers data dependencies and control dependencies from them. As

Attack Surface	Pre-Processing				Dependency Collection						Dependency Combination		
	Tracer LoC	Trace Size	# of APIs	Time (min)	R1	R2	R3	# of Data Deps	# of Control Deps	Time (min)	Initial Score (EFF/DIV/COMP)	Final Score (EFF/DIV/COMP)	Time (min)
`Image`	26,775	2.90 GB	540	125	870	840	232	56,632	(124+0)/(124+5)	1035	23,211/ 5.30/ 1.07	32,795/ 43.30/ 1.06	1,075
`Font`	33,904	7.70 GB	689	180	16,556	1,350	320	192,388	(60+0)/(60+5)	1643	18,782/ 7.10/ 1.06	26,391/ 33.60/ 1.16	534
`PDF`	29,356	1.60 GB	595	95	908	905	233	66,689	(117+0)/(117+6)	371	13,214/ 6.00/ 0.98	19,080/ 43.50/ 1.04	484
`Audio`	18,822	0.13 GB	345	58	107	116	32	11,422	(2+68)/(2+68)	89	11,603/ 6.50/ 1.06	13,061/ 92.00/ 1.06	857
`RTF`	10,442	0.41 GB	191	15	40	40	24	1,396	(30+0)/(30+0)	25	43,721/ 3.00/ 1.00	45,001/ 13.40/ 0.96	723

Table 2: Intermediate results for the whole process of fuzz driver generation. "Trace Size" column represents the total size for all consumer programs, "R1"/"R2"/"R3" columns are the number of data dependencies inferred using the rules in Section 3.1.1, "Initial Score"/"Final Score" columns are averaged objective scores of the fuzz drivers in the first Pareto frontier.

shown in Table 2, the sixth to eighth columns present the amount of inferred relations using **R1**, **R2**, **R3** discussed in Section 3.1.1. The ninth and tenth columns show the two kinds of dependencies finally collected. For the tenth column, the number of control dependencies are represented in the form of (A+B)/(C+D), where A, C are the recognized and total pointer error handling cases inside the traced API functions, B, D are the recognized and total integer error handling cases respectively. The total numbers of error handling cases are counted manually by analyzing all the API functions inside the traces. The eleventh column lists the time for the dependency inference and collection. There are two parts that take most of the running time: the inference of **R3** and the extraction of data dependency. Both of them need to iterate all traces and use algorithms to match the dependencies. Their time costs vary among attack surfaces, e.g., for inference of **R3**, its cost ranges from 4 minutes (`RTF`) to 643 minutes (`Font`). Note that the data dependencies collected vary from thousands to hundreds of thousands for different attack surfaces. We can observe that most of the attack surfaces can contain more than tens of thousands of dependencies, this indicates that *the dependency combination often faces a broad search space in real-world complicated targets*.

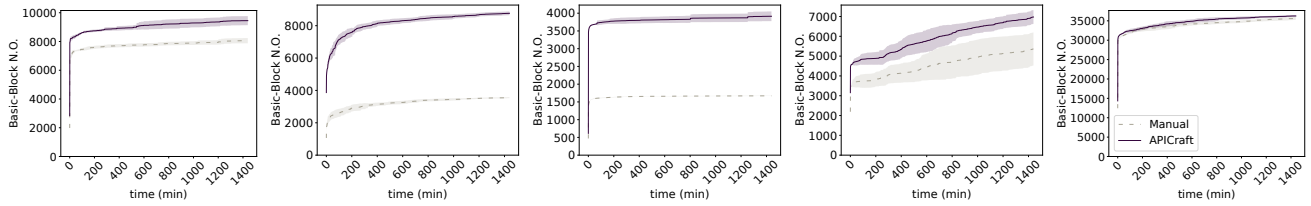
The final step is dependency combination. We ran 300 rounds of evolution and generated 300 new chromosomes for each round. We use this empirical configuration as we found the evolution can converge in 300 rounds for all attack surfaces. The plots for all objective scores during the evolution can be found in Appendix C. The execution time of the genetic algorithm varies from around eight to eighteen hours. The difference of the running time among different attack surfaces is caused by the difference of their compilation time. From what we observed, compilation of the new generated fuzz drivers in the sequentialization step takes most of the combination time. The third-to-last and second-to-last columns show the score before and after the combination. It is the average score of all fuzz drivers in the first Pareto frontier. Note that the EFF and DIV have increased significantly after evolution. This means that the fuzz driver contains more API functions and reaches more basic blocks, calls, and loops to the target library. For some attack surfaces such as `Image` and `RTF`, their COMP scores drop a little after the combination. This phenomenon is reasonable since keeping the fuzz driver as compact as the initial one becomes harder when more data dependencies have been used. It also reflects the conflicts among these objectives. In summary, after evolution, *the fuzz driver has been improved significantly in the desired properties we identified*.

5.2 Comparison with Manually Written Fuzz Driver

To demonstrate the performance of the fuzz driver generated by APICRAFT, we conducted fuzzing experiments to compare the generated fuzz drivers and the manually written fuzz drivers. The experiment setup and generated fuzz driver selection is described in the **Fuzzing Experiment Setting** in Section 5. The manually written fuzz drivers are either collected from the Internet or written by our security analysts. Specifically, for `Image`, the driver is from Project Zero's public repository [32]. For the rest attack surfaces, our security experts write them following the criteria: ❶ the writer has no a priori knowledge of the target attack surface; ❷ each fuzz driver is created in three working days (including the API learning process); ❸ each fuzz driver contains at least one parsing function and one function using the parsing result.

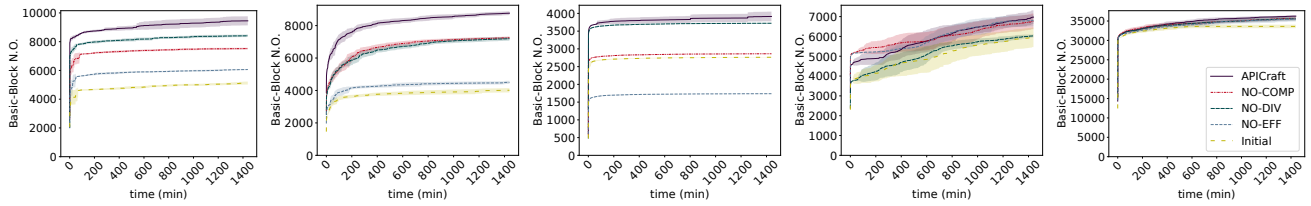
Fig. 7/Fig. 9 shows the coverage/crash comparison results. In Fig. 7, the solid purple line represents the coverage of the generated fuzz driver while the dashed grey line stands for the coverage of the manual fuzz driver. As supplementary, the total basic blocks numbers are `254,680(Font)`/ `413,481(Image)`/ `174,961(PDF)`/ `266,138 (Audio)`/ `418,998(RTF)`. The shadows along the lines are 95% confidence intervals. Based on Mann-Whitney U-test [33], the p-values are `9.13e-5(Font)`/ `9.13e-5(Image)`/ `8.98e-5(PDF)`/ `1.10e-3(Audio)`/ `1.09e-1(RTF)`. All p-values except `RTF` are smaller than `5.00e-2`, which shows the statistical significance. In Fig. 9, only `Audio` and `Font`'s crash results are presented since neither fuzz drivers found any crash in 24 hours on the rest attack surfaces. The p-values are `4.52e-3`/`8.50e-3` on `Audio`/`Font` respectively. Both p-values are smaller than `5.00e-2` showing the statistical significance.

These results show that the generated fuzz driver significantly outperforms the manual written counterpart in most



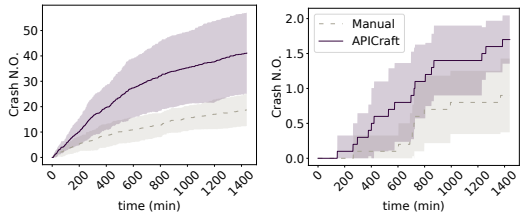
(a) `Image` (b) `Font` (c) `PDF` (d) `Audio` (e) `RTF`

Figure 7: Basic block coverage per time for both APICRAFT generated and manually written fuzz drivers for the five attack surfaces



(a) `Image` (b) `Font` (c) `PDF` (d) `Audio` (e) `RTF`

Figure 8: Basic block coverage per time for both APICRAFT generated with three objectives (APICRAFT), without COMP (NO-COMP), without DIV (NO-DIV), without EFF (NO-EFF), and initial (Initial) fuzz drivers for the five attack surfaces



(a) `Audio` (b) `Font`

Figure 9: Unique crash per time for both APICRAFT generated and manually written fuzz drivers for `Audio` and `Font` attack surfaces (The rest three attack surfaces have found no crash in 24 hours)

	`Image`	`Font`	`PDF`	`Audio`	`RTF`	Avg. Pct.
R1	33/110	55/69	23/87	63/137	12/28	43%
R2	33/110	8/69	24/87	65/137	12/28	32%
R3	32/110	6/69	23/87	15/137	3/28	18%
R*	66/110	64/69	44/87	79/137	15/28	62%

Table 3: Statistics of the used data dependencies inferred from **R1/R2/R3** in final round fuzz drivers. **R*** stands for **R1**, **R2**, and **R3**. For the data in the form of xx/yy, it stands for the number of inferred/total unique dependencies appeared in residents respectively.

cases. However, this doesn't mean that the generated fuzz driver is fully superior than its manual written counterparts. Indeed, the generated fuzz driver wins in its numerous API function calls. For example, the generated driver for `Image` uses 47 different API functions while its manual counterpart only uses 11. By investigating both drivers' code, we found that both types of fuzz drivers have their own advantages. Manually written fuzz drivers are smaller, more concise, easier to understand. Conversely, fuzz drivers generated by APICRAFT behave well in scalability, automation, and they usually contain much more API function calls. More discussion about their relations are in Section 6.1.

5.3 Effectiveness of Each Component

Inference Rules To show the contribution of **R1/R2/R3** to the fuzz driver generation, we collected the number of unique inferred dependencies and all unique dependencies appeared in the final round's residents. Table 3 lists the statistics. On average, **R1/R2/R3** contributes 43%/32%/18% respectively and 62% when combined(the dependencies inferred from **R1/R2/R3** may overlap). *The results show that, though the percentage of the whole amount of inferred dependencies in all extracted dependencies is not high (column 6-8 in Table 2), they significantly contribute to the final fuzz driver generation.*

Ablation Study of Each Objective We conduct fuzzing experiments to understand the contribution of each objective (EFF/DIV/COMP) designed in APICRAFT. For each attack surface, we compare the coverage for five types of generated fuzz drivers including the fuzz driver generated with all three objectives, without COMP, without DIV, without EFF, and the unevolved initial fuzz drivers. Experiment setup and fuzz driver selection follow the setting mentioned at the beginning of Section 5. Fig. 8 depicts the comparison.

By comparing APICRAFT and Initial in Fig. 8, we can draw the conclusion that our algorithm significantly improves the fuzzing drivers (avg. 53% more coverage). Based on Mann-Whitney U-test [33], the p-

Attack Surface	CVE/Issue-ID	macOS Version	Bug Type	Reproduced Apps							
				Messages	Preview	qlmanage	Photo	Safari	Font Book	afclip	QuickTime Player
Image	731907746(R)	10.15.3	FPE	✓	✓	✓	✓	-	-	-	-
	CVE-2020-9790	10.15.4	OOB-Write	✓	✓	✓	✓	-	-	-	-
	CVE-2020-9879	10.15.4	ARB-Write	✓	✓	✓	-	-	-	-	-
	CVE-2020-9961	10.15.5	OOB-Write(Bus Error)	✓	✓	✗	✓	✗	-	-	-
	748048999(U)	10.15.7	ARB-Write	✓	✓	✓	✗	-	-	-	-
	CVE-2021-1793	11.0.1	OOB-Read	✗	✗	✗	✓	-	-	-	-
	CVE-2021-1783	11.0.1	OOB-Write	✓	✓	✓	✗	-	-	-	-
	CVE-2021-1746	11.0.1	ARB-Write	✓	✓	✓	-	-	-	-	-
	756409604(R)	11.1	NPD	✓	✓	✓	✓	-	-	-	-
Font	CVE-2020-9980	10.15.5	OOB-Read	-	-	✓	-	✓	✓	-	-
	737046948(C)	10.15.5	OOB-Write	-	-	✓	-	✓	✓	-	-
	737048356(R)	10.15.5	Stack-Exhaustion	-	-	✓	-	✓	✓	-	-
	738918010(C)	10.15.5	OOB-Read	-	-	✓	-	✓	✓	-	-
	748050615(C)	10.15.7	NPD	-	-	✓	-	✓	✓	-	-
	756641529(C)	11.1	OOB-Read	-	-	✓	-	✗	✗	-	-
PDF	738375428(R)	10.15.5	Stack-Exhaustion	✓	✓	✓	-	✓	-	-	-
Audio	736230948(R)	10.15.5	Infinite-Loop	✗	-	-	-	✓	-	✓	✗
	736230948(R)	10.15.5	NPD	✓	-	-	-	✓	-	✗	✓
	CVE-2020-9866	10.15.5	OOB-Read	✗	-	-	-	✗	-	✗	✗
	CVE-2020-9889	10.15.5	OOB-Write	✓	-	-	-	✓	-	✓	✓
	CVE-2020-27908	10.15.6	OOB-Read	✗	-	-	-	-	-	✓	✗
	744117458(U)	10.15.6	Signed-To-Unsigned-Type-Cast	✗	-	-	-	✓	-	✗	✗
	CVE-2020-9954	10.15.6	OOB-Write	✓	-	-	-	✓	-	✓	✓
	754449272(U)	11.0.1	Interger-Truncation	✓	-	-	-	-	-	✓	✓
	759505458(U)	11.1	Type-Confusion	✓	-	-	-	✓	-	✓	✓
	CVE-2021-1747	11.1	ARB-Write	✓	-	-	-	✓	-	✓	✓

Table 4: Selected macOS bugs detected by APICRAFT. `CVE/Issue-ID`: `R` means this issue is confirmed by vendor but without CVE assigned, `C` means this issue is confirmed by vendor and will assign CVE in the upcoming security update announcement, `U` means this issue is under reviewed by vendor. `Bug Type`: `FPE`: Float-Pointer-Exception, `NPD`: Null-Pointer-Dereference, `ARB-Write`: Arbitrary-Address-Write. `Reproduced Apps`: ✓: reproducible in this app, ✗: non-reproducible in this app, -: the file format of this issue is not supported by this app.

values are `9.13e-5(Font)`/ `9.13e-5(Image)`/ `9.03e-5(PDF)`/ `2.90e-3(Audio)`/ `9.13e-5(RTF)`. All p-values are smaller than `5.00e-2` showing the statistical significance. Besides, according to the scores of first and last round (third-to-last, second-to-last column in Table 2), we observe *a positive correlation between the objective scores and fuzzing coverage*.

By comparing APICRAFT with NO-COMP, NO-DIV, NO-EFF in Fig. 8, we observe that, in general, removing any objective will cause performance drop. The performance drop is more obvious and significant for the attack surfaces which have more collected dependencies, e.g., `Image`, `Font`, and `PDF`. Specifically, more collected dependencies will bring a larger search space to combination and possibly a more complicated dependency graph, which makes the generation harder. *Consequently, given a harder target,* APICRAFT*'s three-objective algorithm can find a better solution (a fuzz driver with higher coverage) than any of the above two-objective algorithms*.

Interestingly, in Fig. 8, the rank for these three two-objective algorithms varies in each attack surface. A possible explanation is that each objective can provide its own feedback during the evolution. Missing one objective will cause the evolution fails to keep certain combinations, i.e., fails to figure out certain parts of the complete solution. Therefore, given an attack surface, the fuzzing performance will drop when missing one specific objective. But the drop rate is specific to each attack surface. Another observation is that evolution without considering the fuzz driver's compactness (NO-COMP) cannot even always get a second-best rank. *This demonstrates that, to generate a better fuzz driver, it is necessary to keep the fuzz driver as simple as possible during the evolution while enriching the program behavior it can trigger at the same time.* We designed APICRAFT towards this goal and proposed a multi-objective solution. Besides, non-coverage metrics can also be discussed. For NO-COMP, we observe that its generated fuzz drivers are longer (e.g., for `Image`, its LoC is more than 5,000 lines while others are less than 1,500). This shows that compactness objective not only helps to improve the quality of generated fuzz drivers but also makes them more maintainable and understandable.

5.4 Fuzzing Campaign

General Results We setup a long-term fuzzing campaign to fuzz these five attack surfaces. The campaign uses all generated fuzz drivers which have unique data dependencies from APICRAFT's output (aka the first Pareto frontier of the final round's residents). So far, 142 unique vulnerabilities have been found. Specifically, 54 of them have been confirmed by Apple and assigned with CVE numbers, and 16 of them have been confirmed with Apple and will be assigned with CVE numbers on the upcoming Apple's security update, 56 of them are still under Apple's reviewing, and the last 16 of them

```
1 CTFontDescriptorRef desc;
2 const CGGlyph* glyphs;
3 int status = 1;
4 ... // the creation of desc from input file is omitted
5 CTFontRef fontRef = createWithFontDescriptor(desc, CONST_DUMP1, NULL);
6 status = getGlyphsForChars(fontRef, CONST_DUMP2, glyphs, CONST_DUMP3);
7 if (status != 0) exit(1);
8 // the vulnerable function
9 getAdvancesForGlyphs(fontRef, CONST_DUMP4, glyphs, NULL, CONST_DUMP3);
```

Figure 10: Minimal fuzz driver for Issue 756641529. (`CONST_DUMP*`: variables that are dumped constant values from the trace.)

are recognized as DoS with no CVE number assigned due to its low security threats. Among all the 142 vulnerabilities, 126 of them are memory corruption vulnerabilities which are possibly exploitable. By manually analyzing the bugs, they are divided into 12 types of root cause, including heap out-of-bound read/write, integer truncation etc. Table 4 lists the selected vulnerabilities and the full list is in Appendix E. Table 4 also shows the affected applications such as `Safari`, `Preview`, etc. Although these vulnerabilities are detected in macOS SDK, indeed they influence the whole Apple ecosystem including macOS, IOS, watchOS, tvOS etc.

Case Study 1: Issue 756641529 Instead of discussing every vulnerability in detail, here we use the vulnerability found in the `Font` attack surface (Issue 756641529 in Table 4) as a representative case for study. This case is small but complete, which can help to demonstrate most features of APICRAFT. Fig. 10 shows the minimal fuzz driver to reproduce issue 756641529 and it is carved from the fuzz driver generated by APICRAFT. For conciseness and easier comprehension, the function names are simplified and each variable is given a meaningful name. (The variable names in the generated fuzz driver are not human-friendly.)

The flow of triggering the vulnerability is as follows: ❶ `createWithFontDescriptor` parses the content of the input font file and return the parsing result as `fontRef`. ❷ `getGlyphsForChars` uses the parsing result (`fontRef`) to fill up glyph information, which is stored in `glyphs`. If no error happens during this step, the program will continue execution. ❸ `getAdvancesForGlyphs` gets the advance information for the glyphs and an OOB-read error can happen if the input font file is malformed.

Here are some key observations from this case: ❶ Only parsing the font is not enough for triggering this vulnerability. Multiple API functions need to be combined together to reach the vulnerable code. ❷ `createWithFontDescriptor` and `getGlyphsForChars` are connected by the data flow of `fontRef`, which is captured by the *extraction* strategy (§ 3.1.1). ❸ `getGlyphsForChars` and `getAdvancesForGlyphs` are connected by the data flow of `glyphs`, which can be captured by the *inference* strategy (§ 3.1.1). ❹ The error code handling for `getGlyphsForChars` is captured as the control dependency (§ 3.1.2). This guarding check helps to eliminate a large portion of false-positive crashes due to API function misuse.

In summary, the case demonstrates that the *collect – combine* approach of APICRAFT helps to build semantically meaningful fuzz drivers to facilitate vulnerability detection.

Case Study 2: ExtAudio API Family For the `audio` attack surface, we found that the generated fuzz drivers can be categorized into two sets based on whether they include functions with the prefix `ExtAudio` in name or not. In other words, most of the functions' names start with `ExtAudio` in some fuzz drivers, while there's no such functions in the others.

We further investigated this issue and found that the audio attack surface involves two sets of independent services: the Audio File Services [34] and the Extended Audio File Services [35]. Both of them have their own APIs for creating the file stub as well as input parsing. As a result, the fuzz drivers generated for these two services are totally different.

In the experiments in Section 5.2, we found that the security analyst only wrote the fuzz driver for the Audio File Services, so we chose the corresponding fuzz driver from the generated suite for fair comparison. Nevertheless, in the long-term fuzzing campaign, we use both types of generated fuzz drivers for fuzzing. In fact, we found that the fuzz drivers involving the Extended Audio File Services contribute a lot of CVEs such as CVE-2020-9866, CVE-2020-9890, CVE-2020-2790 and CVE-2021-1747 from table 4. This is because the Extended Audio File Services include the logic for not only parsing of an audio file but also the decoding of it.

This case study shows APICRAFT indeed helps to unveil more program behaviors which can lead to more bugs.

6 Discussion & Future Work

6.1 Discussion

Human Efforts in APICRAFT Human efforts are inevitable in fuzz driver generation [7, 8, 36]. In APICRAFT, there are several tasks that need human intervention (more discussion in Appendix D). ❶ As discussed in Section 3.2.2, we need manually configure basic knowledge for complementing some data dependencies that cannot be collected by tracing the consumer programs. In the current implementation, all macOS targets share one basic knowledge base and it is encoded as `toml` configurations. ❷ The generated fuzz drivers may contain false positives. False positive means the root cause of a crash found by fuzzing is the misuse of the API functions rather than a bug in the library. Identifying such cases requires manual analysis and domain knowledge. In APICRAFT, the error handling and stability test can significantly eliminate false positives. Nevertheless, we still found one false positive in the generated fuzz drivers for the RTF attack surface. We manually fixed that false positive.

Relation with Manually Written Fuzz Drivers In Section 5.2, we compare APICRAFT with manually written fuzz drivers. Although the experiment results demonstrate APICRAFT's superiority, we still have to admit that APICRAFT cannot totally replace human experts in fuzz driver generation. The rationale is that the information collected by APICRAFT is entirely from the execution traces of existing consumer programs while human experts can learn knowledge from a lot more data sources including but not limited to documents and online code snippets. In fact, two types of fuzz drivers can mutually benefit each other. On one hand, the manually written fuzz drivers can be used as consumer programs to provide more data and control dependencies for APICRAFT. On the other hand, the fuzz drivers generated by APICRAFT can provide not only candidate fuzz drivers but also insights about the mechanisms of target libraries for the human experts to write better fuzz drivers.

6.2 Limitation & Future Work

Although APICRAFT has shown promising results on finding vulnerabilities in closed-source SDKs, it still has several performance limitations. First, the quality of the generated harness is limited by the quality of the execution trace. Since APICRAFT mainly relies on the execution traces to extract data/control dependencies. Second, currently, APICRAFT only supports the data/control dependencies discussed in Section 3.1.1 and Section 3.1.2. The missing dependencies may lead to both false negatives and false positives. Third, APICRAFT focuses on finding memory corruption related vulnerabilities and cannot find concurrency bugs or logical bugs. Fourth, the current implementation of APICRAFT works only for C or C-style APIs of the SDKs.

More Data/Control Dependencies For the data dependencies, APICRAFT currently focuses on the dependencies between function input and output parameters. We choose to use this type of dependencies because they can be extracted from the execution traces accurately. However, there are other types of data dependencies. For example, two functions can exchange data via `struct`'s members/global variables. Such type of dependencies can possibly be captured via monitoring the functions' memory operations during the tracing process but it can cause huge overhead for the tracing tool and the accuracy of the acquired information can be low considering that the consumer programs are large GUI software. For the control dependencies, APICRAFT currently focuses on the error handling related dependencies. This is because, first, the error handling paths often have clear patterns, e.g., calling `_exit`, `_abort`, making them easy to recognize; second, error handling is important for reducing false positives. Indeed, there are other control dependencies, but collecting all of them accurately requires developing advanced binary analysis techniques, which is not the focus of this tool. So we leave it as future work.

Inter-Argument Relation Inference In some functions, there also exist semantic relations among the arguments. For example, a function can have an array as its first argument and the length of this array as its second argument. Intuitively, having the knowledge of the inter-argument relations is beneficial for building fuzz drivers but the acquisition of this knowledge is challenging. In FUZZGEN [8], the authors designed an approach to infer this kind of relationship with value-set analysis. However, the value-set analysis used by FUZZGEN cannot get adopted in APICRAFT directly since it requires the type information of not only the function arguments but also the function variables, which is clear in source code but not in binary. In the future, we plan to utilize binary-level value-set analysis techniques [37] to further improve the robustness of the fuzz drivers generated by APICRAFT.

Non-C Languages Our current implementation focuses on SDK libraries which provide C or C-style APIs. The difficulties for supporting other languages mainly come from the *collect* stage. In theory, the data-dependency and control-dependency modelled by APICRAFT exist in modern programming languages. However, supporting non-C languages requires more engineering efforts and domain knowledge. For example, Objective-C & Swift heavily rely on their language runtime to support their language properties and adaptation of such languages in APICRAFT's current implementation requires deep understanding about them.

7 Related Work

Fuzz Driver Generation The automatic generation of fuzz drivers is an emerging field of study. Some research advances have been published recently on this topic. FUDGE [7] is a technique to automatically synthesize fuzz drivers for open-source libraries. It extracts candidate fuzz drivers from the consumer programs of a library and then presents them to the human expert to make decisions on which one should be used for fuzzing. Meanwhile, FUZZGEN [8] uses the source code of the consumer programs to learn the correct usage of library functions and build an Abstract API Dependency Graph (A^2DG) with the learned knowledge. Then FUZZGEN can generate fuzz drivers by traversing the A^2DG. The key difference between APICRAFT and FUDGE/FUZZGEN is that APICRAFT targets binary-level libraries while FUDGE and FUZZGEN work on source-level. APICRAFT not only addresses the topic of how to build high-quality fuzz drivers, but also identifies and solves the problems unique to binary-level fuzz driver generation. Apart from FUDGE and FUZZGEN, WINNIE [36] aims to fuzz closed-source libraries on Windows via fuzz driver generation and fast-cloning of processes. The first difference between APICRAFT and WINNIE is that WINNIE involves the improvement of fuzzer efficiency on Windows by developing a similar mechanism to `fork` while APICRAFT focuses on fuzz driver generation. The second difference is how the fuzz drivers are generated. In WINNIE,

fuzz drivers are directly extracted from the execution traces of consumer programs. On the contrary, in APICRAFT, fuzz drivers are synthesized based on the learned relations between API functions, allowing APICRAFT to generate fuzz drivers with better diversities for vulnerability revealing.

Unit-test Generation Unit-test Generation is a closely related area for APICRAFT. The current unit-test generation techniques can be categorized into three categories. ❶ The first type of approach is to carve unit-tests from existing tests. Elbaum et al. [38] proposed an approach to carve unit-tests from the execution traces of system tests while Kampmann and Zeller [39] developed a technique to carve unit-tests from C programs. ❷ The second type of approach is random test generation [40–42]. These techniques use static or dynamic analyses to guide the random generation of unit-tests. ❸ The third type of approach is to use evolutionary algorithms to generate a suite of unit-tests [43, 44]. These techniques focus on driving the entire suite of unit-tests towards predefined goals instead of optimizing a particular unit-test. On one hand, APICRAFT differs from the unit-test generation techniques in many aspects such as goals, approaches etc. On the other hand, some of the concepts used in these unit-test generation techniques inspire the design of APICRAFT, for example, the usage of evolutionary algorithms for test generation.

Advanced Fuzzing Techniques Fuzzing has become a well-recognized vulnerability detection technique since first introduced [1]. A lot of research efforts have been devoted to improving both the efficiency and effectiveness of fuzzers in recent years [45–63]. These techniques are orthogonal to APICRAFT since the fuzz drivers generated by APICRAFT can be supplied to any fuzzer.

8 Conclusion

In this paper, we propose APICRAFT, a novel technique for fuzz driver generation for closed-source SDK libraries. The key strategy of APICRAFT is called *collect–combine*. First, APICRAFT collects the dependencies of API functions. Then, it combines the collected dependencies with a multi-objective genetic algorithm to build semantically meaningful and diverse fuzz drivers. Through the evaluation, APICRAFT demonstrates great superiority and capability. Moreover, we have discovered 142 vulnerabilities in macOS SDK with APICRAFT and 54 of them are assigned with CVE IDs.

9 Acknowledgments

We thank our shepherd William Robertson and the anonymous reviewers for their insightful comments on our work. This research is supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, Ant Group through Ant Research Program. The research of Dr Xue is supported by the National Natural Science Foundation of China (Grant No. 61972373) and CAS Pioneer Hundred Talents Program.

References

[1] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities.

[2] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

[3] M. Zalewski. american fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl.

[4] libFuzzer. https://bit.ly/3uS2Uu8.

[5] Honggfuzz. https://bit.ly/3fa4fFG.

[6] Clusterfuzz. https://bit.ly/3hpXimI.

[7] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[8] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. Fuzzgen: Automatic fuzzer generation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[9] Apicraft web page. https://sites.google.com/view/0xlib-harness.

[10] Output parameters. https://bit.ly/3fcr0ce.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii.

[12] R. P. Grimaldi and RoseHulman. *Discrete and Combinatorial Mathematics; An Applied Introduction*.

[13] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*.

[14] Pareto frontier. https://stanford.io/3fiTSQ2.

[15] T. J. McCabe. A complexity measure.

[16] AddressSanitizer. https://bit.ly/3fkZoBG.

[17] libgmalloc. https://apple.co/3hr3vPw.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation.

[19] O. A. V. Ravnås. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re.

[20] Fuzzing ImageIO. https://bit.ly/2SQ2S7R.

[21] Mozilla FuzzData. https://bit.ly/3p765vz.

[22] Imagetestsuite. https://cutt.ly/LbJ9sf9.

[23] Go fuzz Corpus. https://cutt.ly/CbJ3GaU.

[24] AFL Image Corpus. https://bit.ly/3fxCicv.

[25] Fuzzing-project corpus. https://cutt.ly/fbJ3Y8J.

[26] Strongcourage corpus. https://bit.ly/3bqH1u6.

[27] Strongcourage PoCs. https://bit.ly/3eMmwtQ.

[28] Jaanus Kääp's Corpus. https://foxhex0ne.com.

[29] fuzzbench-data. https://bit.ly/3fgmCsJ.

[30] Honggfuzz Corpus Minimization. https://bit.ly/3y5dkZn.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[32] Image Harness. https://bit.ly/3fkYuVZ.

[33] N. Nachar et al. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution.

[34] Audio file services. https://apple.co/33J2jyW.

[35] Extended audio file services. https://apple.co/33HTcyw.

[36] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning.

[37] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*.

[38] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases.

[39] A. Kampmann and A. Zeller. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.

[40] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*.

[41] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs.

[42] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*.

[43] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*.

[44] M. Vivanti, A. Mis, A. Gorla, and G. Fraser. Search-based data-flow test generation. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*.

[45] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain.

[46] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*.

[47] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[48] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[49] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu. Fot: A versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[50] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[51] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[52] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.

[53] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*.

[54] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*.

[55] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*.

[56] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*.

[57] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*.

[58] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[59] X. Xie, L. Ma, F. JuefeiXu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[60] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[61] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[62] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*.

[63] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

A Inter-thread Data Dependency Extraction

Algorithm 3 Inter-thread Data Dependency Inference

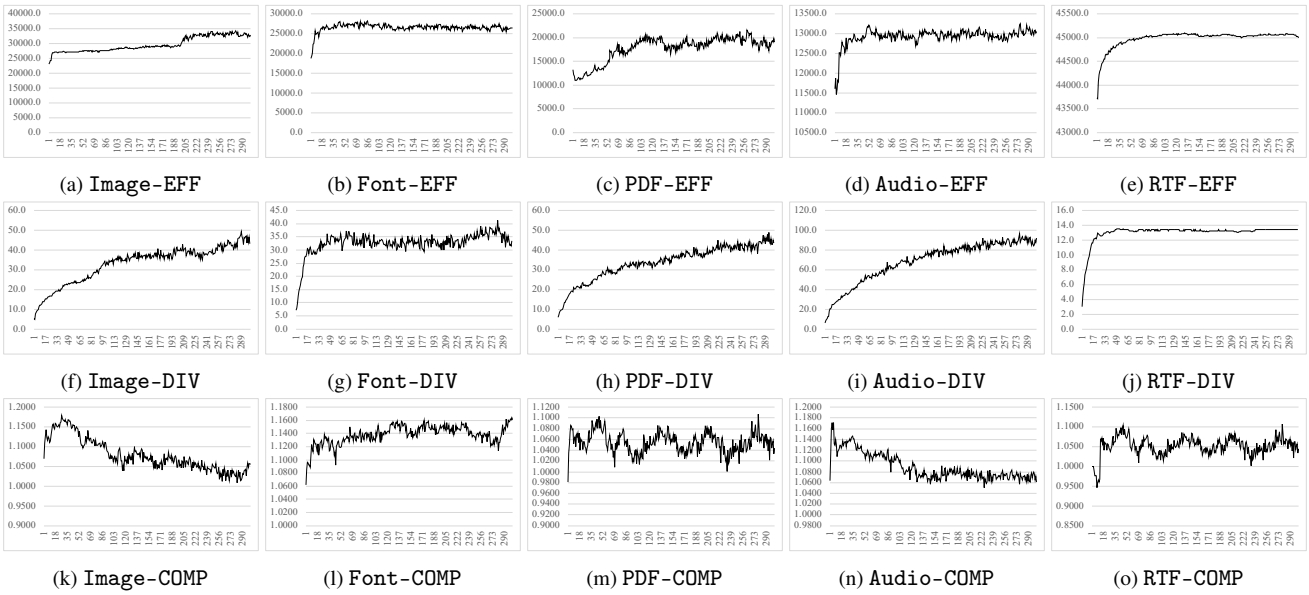
Input: T_1, T_2 (API function traces from different threads)
Output: R (Inter-thread Data dependency set)

```
1:  R ← ∅
2:  cache ← { }
3:  idx ← 0
4:  for F_B ∈ T_2
5:      for F_A ∈ T_1[idx : ]
6:          if F_A ≻ (Exec Order) F_B
7:              idx ← index of F_A in T_1
8:              break
9:          for Out ∈ O_{F_A}
10:             if Out.type ≙ (type) Pointer
11:                 cache ←+ {Out.value : ⟨F_A, Out⟩ }
12:     for In ∈ I_{F_B}
13:         for ⟨F_A, Out⟩ ∈ cache[In.value]
14:             if Out.type ≙ (type) In.type
15:                 R ←+ ⟨F_A, Out, F_B, In⟩
```

The Algorithm 3 shows a simplified process of inferring inter-thread data dependencies (**R3** in Section 3.1.1). Inputs T_1, T_2 are two traces which belong to the same application but different threads, output R is the set of inferred dependencies that the functions in T_1 provide input data of the functions in T_2. This algorithm is modified from Algorithm 1. The key modifications are: ❶ F_A and F_B in this algorithm are iterated from two threads T_1 and T_2. Consequently, in each iteration of F_B (line 4), the *cache* stores all T_1's functions executed earlier than F_B (aka smaller execution order). ❷ This algorithm only finds data dependencies of pointer types or types that are convertible to pointer types (e.g., `int_64` in 64 bit OS). Line 10 combining line 14 shows this.

B Mappings between the attack surfaces and system libraries

Table 5 shows part of the many-to-many relationship between the attack surfaces and the system libraries in macOS SDK.



(a) Image-EFF (b) Font-EFF (c) PDF-EFF (d) Audio-EFF (e) RTF-EFF

(f) Image-DIV (g) Font-DIV (h) PDF-DIV (i) Audio-DIV (j) RTF-DIV

(k) Image-COMP (l) Font-COMP (m) PDF-COMP (n) Audio-COMP (o) RTF-COMP

Figure 11: EFF/DIV/COMP objective scores per time during the dependency combination for the five attack surfaces. The x-axis is the round of the genetic algorithm, the y-axis is the value of the score.

Attack Surface	System Library									
	ImageIO	CoreGraphics	CoreFoundation	vImage	libate.dylib	libOpenEXR.dylib	AudioToolbox	CoreAudio	CoreText	FontParser
Image	✓	✓	✓	✓	✓	✓	-	-	-	-
Audio	-	-	-	-	-	-	✓	✓	-	-
Font	-	✓	-	-	-	-	-	-	✓	✓
PDF	✓	✓	-	-	-	-	-	-	-	-
RTF	-	✓	✓	-	-	-	-	-	✓	-

Table 5: Part of the mappings between the attack surface and system libraries in macOS SDK

C APICraft's Evolution Detail

Fig. 11 shows the evolution detail (EFF/DIV/COMP score per round). Overall, the evolution tries to find residents that can have higher scores in all three objectives.

D Human Efforts for Fuzz Driver Generation

First, to dump the trace, APICraft requires the user to use the program. The current strategy is that the user should at least explore some basic features of the consumer programs with one or more representative input files.

Second, acquiring the basic knowledge about the target library also requires human efforts. It helps to provide several objects' initialization code which cannot be learnt from the traces. The basic knowledge is either out of the target library's scope, e.g., the objects defined in specific languages' base library, or missed in the traces (since APICraft only hooks the functions from the target library). In current implementation, the knowledge is encoded as a toml file (normally dozens lines of code) and loaded by APICraft before the start of the combination. The complete basic knowledge configuration used in our evaluation is available in [9].

APICraft's fuzz drivers may cause false positive crashes. Differentiating them requires human analysis. APICraft provides a dependency graph for each fuzz driver to facilitate manual debugging. Besides, the suggested fix is to comment that crashed function call and its dependent functions according to the graph, and add that unstable/incorrect dependency into the blacklist of combination's configuration.

E Details of Detected Vulnerabilities

We used `honggfuzz` to fuzz the generated fuzz drivers and additionally enabled `libgmalloc` to reveal more subtle memory corruptions. All found crashes and part of the timeouts are manually analyzed. Consequently, we've found 142 unique vulnerabilities which can be divided into 12 types of root causes. Due to the page limit, the full vulnerability list of `Audio`, `Font`, `PDF` and `Image` attack surfaces is listed in [9]. Note that some vulnerabilities we found share one Apple issue id. This is because we reported them together and Apple only assigns one issue id for each report.