2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678910

BIFF: Practical Binary Fuzzing Framework for Programs of IoT and Mobile Devices

Cen Zhang [§]* Yuekang Li [§]* Hongxu Chen [§] Xiaoxing Luo [‡] Miaohua Li [‡] Anh Quynh Nguyen [§] Yang Liu [§]
[§]*Nanyang Technological University* [‡]*Internal Cyber Security Lab, Huawei Technologies Co., Ltd.*

***Abstract*—Internet-of-things (IoT) or mobile devices are omnipresent in our daily life; the security issues inside them are especially crucial. Greybox fuzzing has been shown effective in detecting vulnerabilities. However, applications in IoT or mobile devices are usually proprietary to specific vendors, fuzzers are required to support binary-only targets. Moreover, since these devices are of heterogeneous architecture, assigned with limited resources, and many testing targets are server-like programs, applying existing fuzzing techniques faces great challenges.**

This paper proposes BIFF, a general-purpose fuzzer that aims to stress these issues. It supports binary-only targets, is general (supports multiple CPU architectures including Intel, ARM, MIPS, and PowerPC), fast (has the lowest runtime overhead compared to existing fuzzers), and flexible (uses a new fuzzing workflow that can fuzz any piece of code inside the target binary). Experiments demonstrate that BIFF has the best performance compared with state-of-the-art binary fuzzers and can fuzz the server-like programs which cannot be fuzzed by the existing fuzzers. Using BIFF, we've found 24 unknown vulnerabilities (including memory corruptions, infinite loops, and infinite recursions) in industrial products.

***Index Terms*—Fuzzing, Dynamic Binary Instrumentation**

I. Introduction

Internet-of-Things (IoT) and mobile devices have been broadly used in our daily life. The security issues of these devices have become crucial and fuzzing is one of the most popular and practical vulnerability detection techniques [1]. In particular, AFL [2], libFuzzer [3], honggFuzz [4] together with their variants have been used to detect thousands of software vulnerabilities. Compared to desktop software, the applications and firmware of these devices are extremely hard to fuzz. We summarize the challenges of fuzzing binary programs of embedded or mobile devices as follows:

C1: Few existing fuzzers have binary-only support. A large number of the fuzzing techniques [3, 5, 6, 7, 8] rely heavily on static analyses of source code to instrument extra information for guidance, which usually cannot be ported to proprietary programs since the analyses on binaries are much less precise than those applied on source code.

C2: Few binary fuzzers are general enough to support different CPU architectures. Most existing binary fuzzers only support a specific CPU, which limits their usage on IoT devices with different CPU architectures. For example, honggfuzz [4] relies on Intel BTS or Intel PT tracing support [9], which means that it cannot be applied on AMD powered systems, let alone the ARM, MIPS, PowerPC architectures which have substantial differences from Intel. Therefore, it is preferable to have one binary fuzzing framework that can support as many CPU architectures as possible. Furthermore, it should also be able to scale to new CPU architectures without much effort.

C3: The runtime overhead of binary instrumentation is usually huge. Existing instrumentation techniques used by the binary fuzzers greatly reduce the overall execution speed of the target. For example, AFL's binary-only instrumentation [10] relies on QEMU (denoted as AFL-qemu) as the binary emulator, where the execution speed may be decreased to 1/5∼1/2 the speed of AFL's source code instrumentation (denoted as AFL-src). This shortcoming greatly reduces the fuzzing efficiency. Therefore, it is preferable to provide a extremely lightweight instrumentation mechanism to boost the fuzzing efficiency.

C4: Existing fuzzers are not flexible enough to cope with different testing targets, especially the server-like programs[1]. When fuzzing certain functionalities in the target applications, it would be more efficient if we focus on testing only the relevant code rather than the entire application. For example, after analyzing CVE-2019-11932 [11], we found that WhatsApp's GIF decoder logic is an interesting fuzzing target. Ideally, each fuzzing iteration should start from the code where the GIF files are fed into, and end after finishing decoding. However, existing fuzzers can only fuzz the entirety of the target binary but not a specific piece of logic inside the binary. Similar problems also exist for fuzzing other kinds of server-like applications, e.g., parsing logic inside the Apache HTTP server. To sum up, a new fuzzing workflow should be designed to be compatible for fuzzing these kinds of targets.

In this work, we propose BIFF for solving these abovementioned issues. BIFF has the following key features:

1) **Binary-only support.** Given a closed-source binary, BIFF can dynamically instrument the fuzzing logic into its process at runtime, while keeping the execution semantic of the target process intact.
2) **Generality.** BIFF supports most mainstream CPU architectures including Intel x86_32, Intel x86_64, ARM32, ARM64, MIPS32, PowerPC64, etc.
3) **High-performance.** BIFF has lowest runtime overhead compared with state-of-the-art binary-only fuzzers. Besides, it integrates several lightweight fuzzing algorithms to further boost the fuzzing performance [12], [13].

*They have equal contribution, Yuekang Li is also the corresponding author.

[1]Server-like programs are programs like Apache HTTP server, Linux daemon, Apk applications which normally do not terminate by themselves. They have a different lifecycle compared with traditional command line programs.

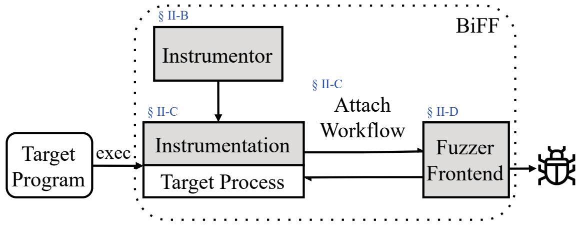


Fig. 1: Overview of BIFF.

4) **Flexibility.** BIFF proposes an attach fuzzing workflow which can fuzz any piece of code inside the target binary while requiring a few configurations.

We applied BIFF on several real world programs. The results show that BIFF not only can be used for server-like programs that existing fuzzers cannot handle, but also achieves the best performance compared with state-of-the-art binary fuzzers. Besides, BIFF has been applied to five series of industrial products of Huawei Technologies Co., Ltd. By far, 24 zero-day bugs have been found, covering various bug types such as memory corruptions, infinite loops, and infinite recursions. All found bugs have been confirmed and fixed by vendor.

II. The BIFF Tool

A. Overview

Figure 1 shows the general workflow of BIFF. The objects labelled with section sign § correspond to the four key components in BIFF's design. First, as a grey-box fuzzer supporting binary-only fuzzing scenarios, BIFF instruments the target program at runtime, which means that it will modify the assembly instructions loaded in the process's memory to inject the instrumentation while keeping the program's logic intact. We developed a dynamic instrumentation engine called Skorpio as the instrumentor (§II-B). The instrumentation code injected into the target process can help control the three key parts of a fuzzing iteration: start, feedback, and end (§II-C). After instrumentation, the target process is ready to accept fuzzing requests from the fuzzer frontend. The fuzzer frontend will communicate with the instrumentation code inside the target to conduct fuzzing (we call this process as attach workflow §II-C). The fuzzer frontend is modified based on AFL (§II-D). In the following, we will detail each key component and how they contribute to solve the above challenges.

B. Dynamic Instrumentation Engine – Skorpio

The key technique used in Skorpio is *dynamic binary rewriting based hooking* [14]. As show in Figure 2, the basic idea is: after the target program is loaded into memory, but before being executed, Skorpio will hijack the program's execution by rewriting the assembly instruction at the hooking point with a jump instruction. This jump instruction points to the instrumentation code which not only carries the customized injection logic but also maintains the consistency of the register contexts before and after executing the injected code. Essentially, the mechanism of hooking ensures **binary-only support** and **high-performance**. The cost of this kind of

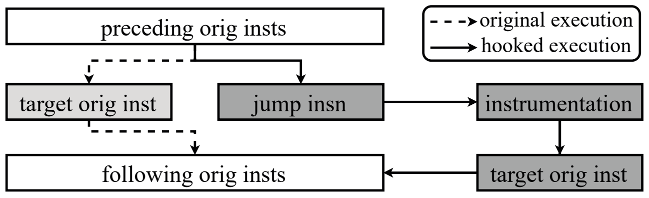


Fig. 2: The basic hooking mechanism.

hooking is minimized and its speed advantage comes from two aspects: ❶ both the original program and the injected code are executed directly on the host machine, unlike the fuzzers that leverage emulation for code instrumentation [10, 4]; ❷ the instrumentation overhead is minimized by using least instructions to accomplish the hooking. Moreover, Skorpio is designed to support binaries of multiple CPU architectures to ensure the **generality** of BIFF. Specifically, this feature is implemented by preparing assembly instruction templates for each supported architecture. Therefore, BIFF can easily support new architectures by adding more templates.

Despite the advantages, using dynamic rewriting based hooking also introduces challenges. The major challenge is that, for variable length instruction sets (CISC, e.g., Intel X86_32), the available hooking points inside the target binary is limited by the distance from the hooking point to the end of the basic block it belongs to. The reason is, if the distance is shorter than the length of the jump instruction, the next basic block's beginning content will be overwritten (broken). And if there are other predecessor basic blocks (not the one contains hooking points), the program logic's intactness cannot be guaranteed. To mitigate this, we design an algorithm using `mmap` to find the closest memory page that the hooking points can jump to, which minimizes the length of the jump instruction. This algorithm significantly increases the number of available hooking points.

C. Fuzzing's Abstraction & Workflow

BIFF abstracts each fuzzing iteration as three key parts: the start, feedback, and the end. Correspondingly, the instrumentation code of BIFF also can be categorized as these three parts. This abstraction helps to flexibly control which code segment to be fuzzed. ❶ For the feedback, BIFF instruments the first instruction of each basic block to collect the coverage information, which is similar as AFL; ❷ For the start and the end, their instrumentations determine where each fuzzing iteration starts and ends. For AFL and its variants, they use fixed strategies: injecting a forkserver at the first instruction of a program binary's main function as the start instrumentation, and no end instrumentation since they assume the program will terminate by itself. The forkserver is an infinite loop where the loop body will fork out a new instance of the target process every time a fuzzing iteration begins [15]. This accelerates the target execution efficiency of the fuzzer by saving the program start time. BIFF takes a step further than them. For BIFF, users can customize the places of these two parts so that it intrinsically can fuzz any piece of code inside the target binary. To complement this feature, BIFF further introduces a new fuzzing workflow called attach workflow. Figure 3 demonstrates

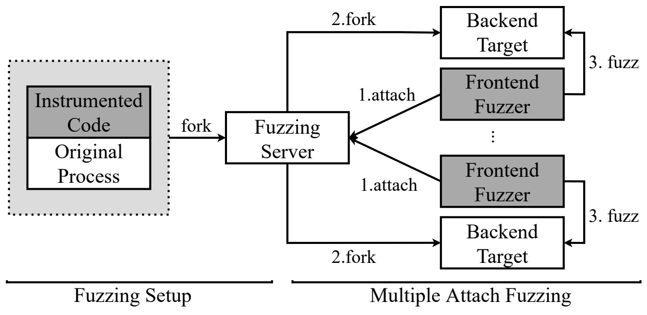


Fig. 3: The attach workflow.

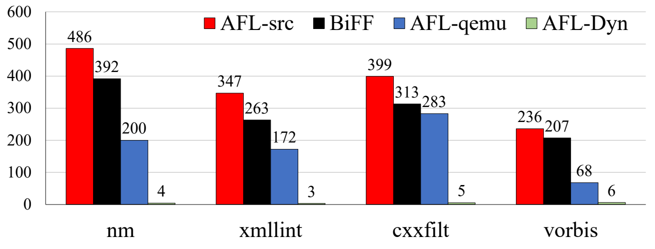
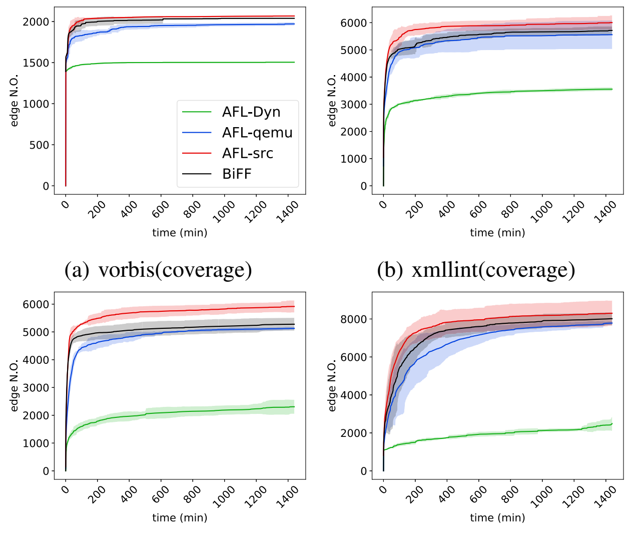


Fig. 4: Fuzzing speed comparison (y-axis: execs/second)



(a) vorbis(coverage)
(b) xmllint(coverage)
(c) cxxfilt(coverage)
(d) nm(coverage)

Fig. 5: Average Coverage Over Time Comparison

this workflow. Unlike existing fuzzing workflow, BIFF attaches to the testing target. The whole workflow is split into two steps, the fuzzing setup step and the multiple attach fuzzing step. In the first step, once the instrumented target process reaches the configured fuzzing start location, it will fork out the fuzzing server process. That fuzzing server loops to wait for the incoming fuzzing request. Next, in second step, after BIFF's frontend fuzzer sends a fuzzing request, the fuzzing server will fork out a backend target process (containing the forkserver logic). The frontend fuzzer will work with that backend target to accomplish the rest fuzzing flow. Besides, multiple attaches is allowed for parallel fuzzing. The attach workflow, together with the three types of instrumentations, ensures the **flexibility** and **high-performance** features.

D. *Fuzzer Frontend*

The fuzzer frontend in BIFF is based on AFL, with three major modifications: ❶ supports attach workflow, which allows the target program to run independently beforehand and wait for the clients to attach to fuzz; ❷ supports fuzzing any piece of code in the target program binary; ❸ integrates advanced fuzzing algorithms [12, 13] for better scheduling and penetration of hard branch condition (e.g., magic byte comparison), which further improves the fuzzer's **high-performance** feature.

III. Implementation & Evaluation

BIFF is implemented as two separate components: ❶ a shared library for instrumenting the target program (built from scratch, 13K LoC in C); ❷ a fuzzer frontend for carrying out whole fuzzing workflow (based on AFL, 14K LoC in C).

A. *Case Study: Fuzzing Native Libraries inside APK*

We choose an Android application as the case study since: ❶ it shows BIFF's generality as no existing fuzzer can directly fuzz native library code inside Android APK (to the best of our knowledge); ❷ it demonstrates BIFF's attach workflow since APK applications are server-like programs and we only fuzz part of its code, i.e., the native libraries which are written in C/C++. Due to the page limit, we put the detail description of the fuzzing process in [16] along with a video demo.

B. *Comparisons With Other Fuzzers*

In total, we evaluated 4 fuzzers, including our tool BIFF, the AFL which requires target's source code (AFL-src), AFL-Dynamorio (AFL-Dyn), and AFL-qemu. The experiment setup follows [17]. Due to the page limit, detailed setting and the crash comparison results are listed in [16].

Fig 4 compares the four fuzzers' average execution speed. AFL-src, which works as the baseline (the compilation stage instrumentation has the lowest overhead), has the best performance. BIFF is fastest among all binary-only fuzzers.

Fig 5 compares the coverage per time for all the four fuzzers. The coverage is collected using AFL's `showmaps` utility. In the plots, lines are average values and shadows along the lines are 95% confidence intervals. BIFF outperforms all other binary-only fuzzers in coverage with statistical significance.

C. *Fuzzing Industrial Products*

BIFF has already been applied to industrial products of Huawei Technologies Co., Ltd. The security researchers of Huawei tested five series of products which include both the traditional command line programs and the server-like programs. Note that this fuzz testing is set as the last step in these products' testing pipeline. Following the confidentiality rules of the company, only part of the statistical results of these bugs are listed. During the fuzzing, `libdislocator` (runtime memory sanitizer [18]) are used if it is applicable. After two man-months of testing (including the time for learning the target programs, setting up fuzzing, fuzzing, and post-fuzzing analysis), 24 unique bugs of the products have been found and confirmed. Specifically, 22 of these bugs are invalid memory read/write issues which may cause security issues. The rest two bugs are infinite loop bug and infinite recursive call bug. We manually analyzed the root causes of all bugs and reported them to the developer team for fixing.

IV. Lessons Learned

Even BIFF has a flexible fuzzing workflow and a practical challenge oriented design, our collaborator (testing team of Huawei Technologies Co., Ltd) still meets various challenges when applying BIFF into industrial products. We discussed and solved all problems together and the following summarizes the lessons we learned.

High configurability and minimum influence on the target program are required for fuzzer to handle intricate fuzzing scenarios. While applying BIFF to different industrial products, we met various obstacles which are usually minor technical issues caused by the implicit execution requirements of the target product. We summarized these requirements into two categories: the target's execution environment requirement and its fuzz configuration requirement. For the execution environment, the target usually has its specific requirements like exclusively occupying some system resources including file descriptors, workspace, process name, etc. For the fuzz configuration, setting up the fuzzing of a target may require more sophisticated tuning of the fuzzer, including when to trigger the start of fuzzing, when to start the runtime sanitizer, where to replace the mutated input, etc. We improve BIFF's configurability from time to time according to the problems we met during the application.

Taking over all I/O functions of the target improves the fuzzing performance. While fuzzing a specific function inside a target program, the testing targets may accept multiple inputs. Specifically, it can read and parse a file in multiple steps or read and parse multiple files. Only mutating the first encountered input content will limit the effectiveness of fuzzing. Here we use a naive but general method to improve the fuzzing performance. We first hook all system level IO operations like `read`, `fread`, etc. Then all read operations get its content from the same place: the input file mutated by the fuzzer. The content for each read operation follows a first-come-first-serve manner, i.e., each read operation will get its content starting from the end location of previous read in the mutated input. The read operation will fail once the input has been totally consumed. This handling strategy helps us find more bugs in the industrial products.

Post-fuzzing analysis benefits both qualitative and quantitative analysis of fuzzing. Currently fuzzing is not a once and for all solution. Its effectiveness requires the users' continuous tuning. During fuzzing, BIFF dumps the executed basic block information to facilitate manual assessment for further adjustment of configuration. These coverage information helps to show whether the fuzzer works properly, whether the fuzzer deviates from exploring the expected targets, which areas are less tested, etc. This helps our human experts to better control the fuzzing process.

V. Future Work

More automatic configuration. Currently, BIFF requires manual configuration, including where to start, stop one testing iteration during the fuzzing. For traditional command line programs, BIFF can automatically set the start and end of the `main` function as the start and end of a testing iteration. However, for the server-like programs, it needs manually configuration. Determining the interesting fuzzing places and identifying fuzzing parameters (input buffer address, length, etc) inside a binary executable requires advanced program analysis techniques. We leave this as our future work.

More advanced fuzzing techniques. BIFF integrates [13] for penetrating magic bytes comparison related branch conditions. In the future, we plan to integrate more effective algorithms to facilitate the fuzzing [7, 5, 19, 20, 21].

VI. Related Work

TABLE I: Fuzzer features Comparison. For the "supported CPUs" column, I: Intel x86_32, Intel x86_64; A: ARM32, ARM64; M: MIPS32; P: PowerPC64. The "Runtime Overhead" column measures the fuzzers' overheads against AFL-src.

	Binary-Only Support	Supported CPUs	Runtime Overhead	Segmentation Support
AFL-src	✗	–	1.0x	–
BIFF	✓	I,A,M,P	1.1x∼2.4x	✓
AFL-pin	✓	I	7.4x∼48.6x	✗
AFL-qemu	✓	I,A,M,P	2.0x∼5.0x	✗
AFL-Dyninst	✓	I,A	1.2x∼3.0x	✓
AFL-Dynamorio	✓	I,A	3.8x∼12.3x	✓
Firm-AFL	✓	I,A,M,P	2.0x∼5.0x	✗

There have been several works that aim to provide binary-only fuzzing. Table I compares these fuzzers with AFL-src as the baseline. The column segmentation support means whether the fuzzer supports a code segment. From Table I, we can see that BIFF is the fastest binary-only fuzzer which supports most types of CPUs with segment fuzzing capability.

VII. Conclusion

This paper presents BIFF, a general-purpose binary fuzzing framework. It supports binary-only fuzzing scenario, most mainstream CPU architectures, has high performance and high flexibility. Its design centers around solving the challenges of fuzzing IoT or mobile applications practically. The experiments shows that BIFF has better performance than state-of-the-arts and can fuzz the server-like programs that others cannot. Till now, BIFF has been used for fuzzing five series of industrial products and found 24 unknown bugs.

Acknowledgment

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.

References

[1] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *CoRR*, 2018.

[2] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2014.

[3] LLVM, "libfuzzer," https://llvm.org/docs/LibFuzzer.html, 2015.

[4] Google, "honggfuzz," https://github.com/google/honggfuzz, 2010.

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

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

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

[8] 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*, 2018, pp. 2095–2108.

[9] Intel, "Intel® 64 and ia-32 architectures software developer's manual," https://intel.ly/3mwFKYT, 2016.

[10] M. Zalewski, "High-performance binary-only instrumentation for afl-fuzz," https://github.com/mirrorer/afl/tree/master/qemu_mode, 2014.

[11] "How a double-free bug in whatsapp turns to rce," https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/, 2019.

[12] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[13] Y. Li, B. Chen, M. Chandramohan, S.-W. 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*, 2017, pp. 627–637.

[14] N. A. Quynh, "Skorpio: Advanced binary instrumentation framework," http://groundx.io/docs/Opcde2018-skorpio.pdf.

[15] M. Zalewski, "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2014.

[16] "Biff website," https://sites.google.com/view/biff-fuzzer.

[17] 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*, 2018, pp. 2123–2138.

[18] "libdislocator, online memory sanitizer," https://github.com/mirrorer/afl/blob/master/libdislocator/README.dislocator.

[19] 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*, 2019, pp. 533–544.

[20] 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)*, 2020, pp. 2325–2342.

[21] C. Zhang, X. Lin, Y. Li, Y. Xue, and Y. Liu, "Apicraft: Fuzz driver generation for closed-source {SDK} libraries," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021, pp. 2811–2828.