Applying Fuzz Driver Generation to Native C/C++ Libraries of OEM Android Framework: Obstacles and Solutions

Shiyan Peng
Fudan University
Shanghai, China
pengsy21@m.fudan.edu.cn

Yuan Zhang
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Jiarun Dai
Fudan University
Shanghai, China
jrdai14@fudan.edu.cn

Yue Gu
Fudan University
Shanghai, China
guy22@m.fudan.edu.cn

Zhuoxiang Shen
Fudan University
Shanghai, China
zxshen22@m.fudan.edu.cn

Jingcheng Liu
Fudan University
Shanghai, China
jingchengliu21@m.fudan.edu.cn

Lin Wang
Fudan University
Shanghai, China
wang_lin@fudan.edu.cn

Yong Chen
OPPO
Chengdu, China
chevin@oppo.com

Yu Qin
OPPO
Chengdu, China
qinyu1@oppo.com

Lei Ai
OPPO
Chengdu, China
ailei@oppo.com

Xianfeng Lu
OPPO
Chengdu, China
luxianfeng@oppo.com

Min Yang
Fudan University
Shanghai, China
m_yang@fudan.edu.cn

ABSTRACT

Fuzz driver generation (FDG) is a fundamental technique for fuzzing library software. Existing FDG approaches have been highly successful with open-source libraries. However, in practice, due to the complex nature of OEM Android frameworks (e.g., customized compilation toolchains, extensive codebases, diverse C/C++ language features), it is not straightforward to integrate existing fuzz driver generation tools with OEM Android libraries. To address this challenge, we first systematically summarize the obstacles to applying existing tools (e.g., FuzzGen) to libraries of an OEM Android (i.e., ColorOS), including compatibility, usability, and effectiveness issues. Following this, we developed a new fuzz driver generation tool, namely FuzzGen++, specifically designed to tackle these obstacles one by one. In our evaluation, we demonstrate the advantages of FuzzGen++ in real-world OEM Android frameworks. FuzzGen++ is compatible with OEM Android and can generate fuzz drivers for all its libraries which are not supported by existing works. The additional analysis of the OEM Android code also enhances its usability within the system. Overall, FuzzGen++ has helped automatically generate 21,457 fuzz drivers. Additionally, through fuzz driver ranking and selection solution, FuzzGen++ figured out cut off 95% fuzz drivers which are less useful. FuzzGen++ supports sophisticated C/C++ features in code analysis, ensuring effectiveness. Compared to hand-written fuzz drivers, FuzzGen++ could generate and select fuzz drivers providing a 107.92% coverage improvement. Furthermore, they discovered 6 bugs, showcasing the capability of FuzzGen++ to find real-world issues.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software security engineering**.

KEYWORDS

Fuzz Driver Generation, OEM Android, Native C/C++ Libraries

ACM Reference Format:
Shiyan Peng, Yuan Zhang, Jiarun Dai, Yue Gu, Zhuoxiang Shen, Jingcheng Liu, Lin Wang, Yong Chen, Yu Qin, Lei Ai, Xianfeng Lu, and Min Yang. 2024. Applying Fuzz Driver Generation to Native C/C++ Libraries of OEM Android Framework: Obstacles and Solutions. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3691620.3695266

1 INTRODUCTION & CHALLENGE OUTLINE

OEM (Original Equipment Manufacturer) Android frameworks refer to customized versions of the Android operating system designed to meet the specific requirements of commercial devices in industrial sectors. The framework employs C/C++ libraries for low-level system resource access [5–8]. Given their frequent operation within high-privilege layers, C/C++ libraries' security is of vital importance [2, 3, 10, 19]. Fuzzing is an outstanding vulnerability detection technique that feeds unexpected, random, or invalid inputs to a program to uncover various vulnerabilities [1, 4, 9, 18, 21]. To apply fuzzing to OEM Android C/C++ libraries, a fuzz driver is necessary to invoke APIs by feeding them with test cases generated by fuzz drivers. Fuzz drivers should be crafted with correct and robust API usage, or it will lead to less productive fuzzing results, which damage the effectiveness of testing. Therefore, crafting fuzz drivers often requires expertise in both OEM Android C/C++ libraries and fuzzing methodologies [17], which is labor intensive.

To address this problem, applying the fuzz driver generation (FDG) technique to OEM Android C/C++ libraries presents a promising solution. Various FDG academic approaches [12–16, 20, 22, 23] have demonstrated great effectiveness in generating usable fuzz drivers for openly accessed libraries. Their fuzz drivers have significantly contributed to finding real-world vulnerabilities in libraries under fuzzing. Among these approaches, consumer-based FDG is mainstream. These approaches generate fuzz drivers from API usage implied in existing code that invokes APIs (i.e. consumer code), making them more likely to avoid API usage mistakes. However, it is non-trivial to migrate these approaches to native C/C++ libraries of OEM Android frameworks. The key reasons are three-fold:

- **Compatibility**: Consumer-based FDG approaches [13, 14, 22, 23] leverage detailed program analysis to extract the semantics of consumer code. Specifically, those designed for C/C++ conduct their analyses based on the LLVM. However, these approaches face compatibility obstacles in OEM Androids, since OEM Androids often employ customized LLVM toolchains that are not fully compatible with the standard LLVM.
- **Usability**: The lack of consumers who provide the usage of APIs makes it difficult to apply FuzzGen to OEM Android C/C++ libraries. This is because OEM libraries are often not well documented and lack usage examples and unit tests, which are commonly served as consumers. Consequently, a comprehensive audit of the OEM Android framework is essential to gather enough consumers. However, the extensive nature of this codebase presents a challenge, as potential consumers are widely dispersed, making it difficult to extract an adequate set of consumers. Existing approaches do not provide a solution for this requirement. Furthermore, once an adequate new set of consumers is identified, a significant number of new fuzz drivers will be generated. It requires much more human-efforts in the fuzz driver selection for effective testing.
- **Effectiveness**: OEM Android C/C++ libraries frequently utilize advanced C++ features, such as virtual functions for polymorphism. Libraries with these features are common, but few existing approaches support them. This limitation hinders their effectiveness because they struggle to extract API usage for APIs for such libraries, often resulting in fuzz drivers with mistakes like incorrect argument passing. Additionally, some libraries are service libraries, which do not have regular consumers that directly invoke APIs in consuming libraries. This makes it challenging for existing approaches to generate fuzz drivers for these libraries.

In this paper, we explore solutions to these obstacles by applying an open-source FDG tool, FuzzGen [14], to OEM Android. FuzzGen was chosen because it is designed for third-party libraries of open-source Android, which closely aligns with our system under fuzzing. However, like other C/C++ FDG approaches, it also suffers from the common obstacles above. As the solution for the obstacles, we introduce a novel fuzz driver framework, named FuzzGen++, specifically developed to overcome the practical obstacles associated with using FuzzGen on the OEM Android (i.e., ColorOS). To address compatibility obstacles, our framework recovers the source code of the OEM LLVM toolchain by utilizing patches and applies these compatibility fixes to FuzzGen++. This enables FuzzGen++ to analyze the code of OEM Android and generate fuzz drivers that can be compiled by the toolchain. In response to usability obstacles, primarily arising from the substantial manual effort needed for collecting consumer code, we have introduced a consumer collection module. To enhance effectiveness, we have introduced more fine-grained data-flow analysis to mitigate inaccuracies.

In our evaluation, FuzzGen++ impressively generated fuzz drivers for 91 libraries. In the fuzzing campaign of these fuzz drivers, we found that they achieved significantly better coverage, with an average improvement of 107.92%, compared to manually crafted fuzz drivers by security experts from the OEM vendor. Additionally, FuzzGen++'s fuzz drivers identified 6 unknown software bugs.

Contributions. This paper makes the following contributions:

- We summarize the typical obstacles in applying existing fuzz driver generation tools on OEM Android C/C++ libraries.
- We propose FuzzGen++, a consumer-based fuzz driver generation approach, which utilize practical solutions for addressing obstacles of compatibility, usability, and effectiveness.
- FuzzGen++ was evaluated on 91 OEM Android framework libraries and generated 1,695 fuzz drivers. These fuzz drivers brought 107.92% coverage improvement compared with manual crafted ones and 6 bug identification.

2 FUZZGEN++

The overview of FuzzGen++ is illustrated in Figure 1. FuzzGen++ first leverages compilation preparation to ensure FuzzGen is compatible with the OEM Android compilation toolchain. Then, to address obstacles on applying FDG to the OEM Android, FuzzGen++ has enhanced and added modules to vanilla FuzzGen. These modules are highlighted in Figure 1. The enhanced modules (illustrated as the grey boxes) means they exist in FuzzGen and have been enhanced in FuzzGen++. The added modules (illustrated as black boxes) are exclusive in FuzzGen++.

2.1 Phase-1: Compilation Preparation

The lack of the LLVM source and the incompatible implementation in FuzzGen prevent FuzzGen from being usable in OEM Android. The Compilation Preparation phase is designed to address these issues and involves two modules. First, the phase Retrievals LLVM with the help of OEM LLVM patches. After obtaining LLVM, we perform manual fixes on compatibility issues within FuzzGen. The details of these two modules will be described below:

OEM LLVM Retrieval. The LLVM used by OEM Android has extensive customized modifications across various sub-projects. Although the complete source code isn't provided, patches for these modifications exist in the LLVM toolchain. These patches, maintained similarly to the official Android toolchain, are based on a specific LLVM mainline version and saved as patch files. The relationship between sub-projects and patches is recorded in a patch entries file. The tool retrieves the LLVM mainline version from

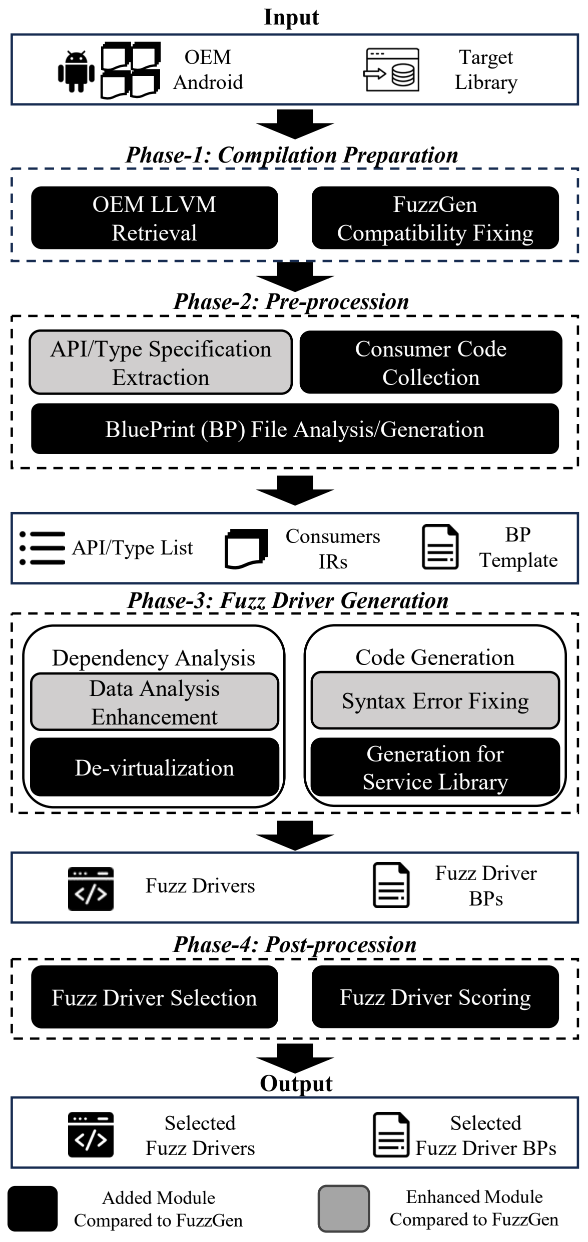

Figure 1: Overview of FuzzGen++.

pre-compiled binaries in the toolchain, then parses the entry file to map and apply the necessary patches.

FuzzGen Compatibility Fixing. Our strategy to update FuzzGen involves four main areas, primarily focusing on replacing outdated APIs with newer alternatives and updating how APIs are used. Specifically, we've replaced deprecated APIs, like `getElementType`, used for type retrieval in data structure dependency analysis, with updated ones such as `getPointerElementType`. Additionally, we've adapted to changes in the latest LLVM version used in OEM Android, which replaces direct string returns with the `StringRef` data structure, by integrating support for extracting the necessary strings from this structure for vanilla FuzzGen.

2.2 Phase-2: Pre-processing

FuzzGen++ takes API and data type definitions, along with consumers, as input. It analyzes the APIs and data dependencies on the consumer to confirm API sequences and data points, generating the fuzz driver's source code. To ensure usability, the tool also requires compilation configuration information, which is handled by a pre-processing phase. This phase is divided into three modules:

API/Type Specification Extraction. The tool extracts linkage names (i.e. mangled) from C++ APIs in IR by compiling the target library into an archive file and using `nm` from `binutil` to extract API symbols. For type analysis, it employs AST analysis to extract data structure definitions within the library, recognizing C++ elements by traversing relevant AST nodes.

Consumer Code Collection. Manual IR collection for consumers in OEM Android is labor-intensive, so an automated analysis step was introduced. This involves collecting IRs and performing API reference analysis across all OEM Android projects. The phase excludes legacy code and uses string-based analysis to identify function call sites in the IR, ensuring precise matching with an API list that includes mangled names. For indirect callsites of the virtual functions, it exploits the LLVM Control Flow Integrity (CFI) mechanism, which provides candidate APIs for such callsites.

Blueprint File Analysis and Generation. To generate Blueprint configuration files, FuzzGen++ analyzes and matches dependencies and compilation options from the target library's Blueprint file. It then arranges and records this information in the Blueprint format, ensuring that the generated fuzz drivers are correctly configured.

2.3 Phase-3: Fuzz Driver Generation

The fuzz driver generation phase, essential to convert consumer code into fuzz driver source code, has two main stages: Dependency Analysis and Code Generation. Dependency Analysis extracts consumers of APIs in library and build control and data flow dependencies within the consumer's LLVM IR. This is crucial for determining the API invocation sequence and argument strategy for the fuzz drivers. Based on the identified dependencies, FuzzGen++ generates the actual code to fuzz the APIs with the help of the enhancements.

De-virtualization. To support virtual function analysis in FuzzGen, FuzzGen++ implements a method within its consumer analysis process. Similarly to consumer collection, it retrieves the API consumed using the information provided by the CFI mechanism. However, in de-virtualization, FuzzGen++ only considers candidates related to the class in a specific library as the API that the call site consumes. This enhancement enables the analysis of virtual APIs and improves the analysis of virtual API calls.

Data Analysis Enhancement. To improve FuzzGen's data modeling, the tool expanded the definition of source points to include arguments, non-API function returns, and non-Alloca defined variables. Additionally, it refined propagation rules to support field-sensitive taint analysis, allowing for more accurate data dependency analysis by considering the fields within data structures.

Generation for Service Library. For OEM Android service libraries, FuzzGen++ builds a specific type of fuzz driver. These fuzz drivers first create a mocked service and then randomly invoke handler APIs within the service. To generate such fuzz drivers, FuzzGen++ extracts the service object APIs and classifies them into construction APIs and regular APIs based on their mangled names For the random invocation of other APIs, FuzzGen++ uses an extended version of `fuzzService` [11], which now supports services in OEM Android for legacy Android systems.

Syntax Error Fixing: We observed that certain statements in the FuzzGen-generated fuzz driver source code, specifically those declaring arrays with the `auto` type, were incompatible with the OEM toolchain. We fix the error by replacing `auto` with a specific type, ensuring compatibility with the OEM toolchain.

2.4 Phase-4: Post-procession

Fuzz Driver Selection. Fuzz drivers could produce nearly meaningless results, such as immediate crashes or frozen coverage. These issues arise from deconstruction mismatches and non-mutation problems. Deconstruction mismatch problems occur when a fuzz driver tries to deconstruct an object that was never constructed. FuzzGen++ identifies this by performing a backward trace to check for prior construction. Non-mutation problems occur when fuzz drivers operate on API sequences lacking arguments suitable for mutation or do not engage significant branches within the API's processing logic. FuzzGen++ detects this by evaluating in-API branches affected by root mutated data. Identifying and excluding fuzz drivers with these issues helps FuzzGen++ improve the efficiency and relevance of the fuzzing process.

Fuzz Driver Scoring. Effective fuzz drivers can explore a broader range of states/paths. Based on the assumption, FuzzGen++ evaluates fuzz drivers by counting API invocations and fuzzed arguments. A higher number of API invocations and fuzzed arguments results in a superior ranking. This scoring helps prioritize which fuzz drivers should undergo testing first. The evaluation process identifies call sites within a fuzz driver, identifies the fuzzed arguments for each API call, and aggregates such arguments to derive a score for each fuzz driver. The details are shown in Algorithm 1.

Algorithm 1 Fuzz Driver Scoring

```
Input: ❶ Source Code of Fuzz Driver $Source_{fd}$; ❷List of APIs in Target Library $APIs$;
Output: Score of Fuzz Driver $Score_{fd}$;
1:  $(Callsites, MutationValues) \leftarrow Source_{fd}$;
2:  for $((Callee, Arguments) \in Callsites)$ do
3:      if $Callee \in APIs$ then
4:          $Score_{Invocations} += 1$;
5:      else
6:          continue;
7:      for $((Arg) \in Arguments)$ do
8:          $Definer_{Arg} \leftarrow (Arg, Source_{fd})$
9:          if $0 < |MutationValues \cap Definer_{Arg}|$ then
10:             $Score_{Arguments} += 1$;
11: $Score_{fd} \leftarrow Score_{Arguments} + Score_{Invocations}$;
```

3 RESULTS

3.1 Experiment Setup

We conduct extensive experiments to evaluate FuzzGen++ on the OEM Android of our industrial collaborator. Specifically, our experiments seek to answer the following research questions:

- *RQ1: General Results of Fuzz Driver Generation.*
- *RQ2: How well FuzzGen++ can address the compatibility obstacles?*
- *RQ3: How well FuzzGen++ can address the usability obstacles?*
- *RQ4: How well FuzzGen++ can address the effectiveness obstacles?*
- *RQ5: How effective FuzzGen++ is in detecting real-world bugs?*

Dataset. To comprehensively evaluate FuzzGen++, we meticulously curated a dataset comprising OEM Android native framework libraries. This dataset is built with two primary criteria: exclusive within the OEM Android and dataset size for thorough assessment.

The dataset construction involved three key steps: Initially, compiling all files from OEM Android's framework directory ensured a diverse library selection. Subsequently, we identified and selected projects compiled into libraries, focusing exclusively on OEM-specific additions distinct from AOSP. This rigorous process resulted in a dataset that included 91 libraries, covering impotent functions such as media decoding, texture parsing, and network management. These libraries are integral to media, network services, and essential components during system boot.

Tool Configuration. Since modifications to the API sequence result in usage patterns that are not present in OEM Android, it is not meaningful to conduct tests on them. Therefore, we do not utilize the API sequence merging functionality of FuzzGen++. Considering real-world testing scenarios, we configured the tool with a 'one library, one day' setting. Specifically, the count of consumers to generate fuzz drivers is set to 20 for a library. For each consumer, we allocated a 10-minute timeout for its generation. The consumer collection process would not exceed 180 minutes, and post-processing takes less than 30 minutes. Under these settings, the entire generation process of one library can be completed within 8 hours.

3.2 RQ1: General Results

We applied FuzzGen++ to all libraries in our dataset. The results are shown in Table 1. In total, it collected **30,153** consumer functions and successfully generated a total of **21,457** fuzz drivers. For every library, it collected consumers and generated fuzz drivers, showcasing FuzzGen++'s capability to address existing challenges. The generation process was completed within a reasonable timeframe, adhering to our specified time constraints. Specifically, FuzzGen++ averaged a fuzz driver generation time of about **5** seconds. Regarding consumer collection, the analysis took less than 8 hours for the **91** libraries. Given that consumer collection is a one-time process, this time cost is tolerable.

After harvesting the fuzz drivers, we used the post-procession to select the top-scored fuzz drivers for each library, limiting to a maximum of 20 per library, resulting in a total of **1,695** selected fuzz drivers. We defined a fuzz driver as runnable if it can survive 5 minutes of fuzzing and achieve at least 5 new state findings. Our results indicate that each library has at least one runnable

Table 1: Overall Results of FuzzGen++

# of Libraries	# of Consumer Functions	# of Generated Fuzz Driver	# of Selected Fuzz Drivers	Average Time Cost for Consumer Collection	Average Time Cost for Fuzz Driver Generation
91	30,153	21,457	1,695	1,534s	1.02s

Table 2: Statistics of Patches Applied

Compiler	LLVM Main Version	Patch	Lines
OEM-Compiler-1	LLVM-11	15	5,078
OEM-Compiler-2		17	5,333
OEM-Compiler-3		20	5,664
OEM-Compiler-4		34	5,354
OEM-Compiler-5		40	5,965
OEM-Compiler-6	LLVM-14	93	34,002
OEM-Compiler-7		72	27,614
Sum	-	291	89,010

fuzz driver. 69.0% (1,169/1,695) of the selected fuzz drivers were runnable, demonstrating the effectiveness of our selection process.

We also compared the performance of manually crafted fuzz drivers with those generated by FuzzGen++, specifically focusing on coverage. We selected 10 libraries containing manually crafted fuzz drivers from our industrial partner and chose the top-scoring driver for each library for comparison. We observed that FuzzGen++-generated drivers achieved significantly higher coverage in a 2-hour fuzzing session across all libraries, with an average coverage improvement of 107.92%. These results demonstrate that FuzzGen++ is capable of generating high-quality fuzz drivers and that its scoring mechanism effectively identifies them.

3.3 RQ2: Adddressing Compatibility Obstacles

While addressing compatibility obstacles, we focused on retrieving the LLVM code of OEM Android, updating the original FuzzGen code, and adding support for blueprint files, which are present in all fuzz drivers. To illustrate the extent of these efforts, we report on the patches applied. As previously mentioned, we used two compilation systems, which together involved seven versions of LLVM based compilers. The details are detailed in Table 2. In total, we applied 291 patches, including approximately 89,010 lines of code. Our tool automates the patching process and helps testers to apply these patches without requiring in-depth knowledge of LLVM. In the second phase, we apply 4,265 lines of code to FuzzGen. In addition to ensuring correctness, we also introduced optimizations to enhance efficiency in API matching.

3.4 RQ3: Addressing Usability Obstacles

The extensive volume of consumers and fuzz drivers highlights a significant challenge for our tool. FuzzGen++ collected tens of thousands of consumers and generated a similar number of fuzz drivers. A manual review validated the accuracy of consumer collection, revealing some false positives in virtual function analysis. This issue occurs because referencing a virtual function does not always correlate with referencing the library API, potentially leading to over-collection of consumers. However, this over-collection's impact is mitigated by a secondary review of call sites during consumer analysis, which is relatively swift. Despite occasional false positives, the manageable analysis time justifies this trade-off.

3.5 RQ4: Addressing Effectiveness Obstacles

Enhancements to support C++ features have positively impacted 79 out of 91 of the libraries, affecting 95.7% (20,524) of fuzz drivers. These drivers now correctly invoke C++ APIs and handle C++ arguments. In comparison, the vanilla FuzzGen would fail to generate fuzz drivers for these libraries due to the absence of the necessary generation schedule and the inability to handle new data flow patterns and source analysis requirements.

3.6 RQ5: Real-world Bugs Detection

To date, FuzzGen++ has successfully identified 6 confirmed bugs within OEM Android, covering a diverse range of issues such as two null pointer dereference (NPD) bugs, two out-of-bounds (OOB) bugs, one memory leak, and a reachable assertion. These findings cover multiple system functionalities, from text and media parsing to service codes operating at elevated privilege levels, demonstrating FuzzGen++'s ability to detect bugs. This underscores its contribution to strengthening the security of crucial system components.

The identification of two NPD bugs and one OOB bug within a service library is attributed to fuzz drivers generated using the service generation mode. This achievement highlights the effectiveness of the service generation mode in improving fuzzing processes.

4 CONCLUSION

This paper explores the obstacles of applying fuzz driver generation to OEM Android C/C++ libraries. To be specific, it identifies 9 obstacles in compatibility, usability, and effectiveness. Then it introduces specific solutions to address these obstacles, such as compatibility with the OEM Android toolchain, newly added consumer collection, and enhanced overall effectiveness of FuzzGen. The experiments show that these solutions are helpful in the creation of effective fuzz drivers and the discovery of real-world bugs.

This study aims to highlight the discrepancy between academic fuzz driver generation methods and the practical complexities of industrial environments. Future efforts can concentrate on improving fuzz driver selection for higher quality, developing a universal solution to bridge compilation and generation gaps, or refining generation techniques for better bug detection.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172105, 62402116). Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] Go-fuzz: randomized testing for go. https://github.com/dvyukov/go-fuzz, 2015.

[2] New critical vulnerability in multiple high-privileged android services. https://www.zimperium.com/blog/cve-2018-9411-new-critical-vulnerability-multiple-high-privileged-android-services/, 2018.

[3] Samsung patches 0-click vulnerability impacting all smartphones sold since 2014. https://www.zdnet.com/article/samsung-patches-0-click-vulnerability-impacting-all-smartphones-sold-since-2014/, 2020.

[4] American fuzzy lop. https://lcamtuf.coredump.cx/afl, 2023.

[5] Android connection (network). https://source.android.com/docs/core/connect, 2023.

[6] Android graphics. https://source.android.com/docs/core/graphics, 2023.

[7] Android media. https://source.android.com/docs/core/media, 2023.

[8] Android native apis. https://developer.android.com/ndk/, 2023.

[9] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2023.

[10] Millions of smartphone users at risk of serious security flaw. https://eu.community.samsung.com/t5/galaxy-s22-series/millions-of-smartphone-users-at-risk-of-serious-security-flaw/td-p/7151183, 2023.

[11] Aidl fuzzing. https://source.android.com/docs/core/architecture/aidl/aidl-fuzzing, 2024.

[12] D. Babi, S. Bucur, Y. Chen, F. Ivani, 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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[13] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[14] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[15] J. Jang and H. K. Kim. FuzzBuilder: Automated building greybox fuzzing environment for C/C++ library. In *ACM International Conference Proceeding Series*, 2019.

[16] J. Jiang, H. Xu, and Y. Zhou. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[17] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[18] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[19] W. R. Vasquez, S. Checkoway, and H. Shacham. The most dangerous codec in the world: Finding and exploiting vulnerabilities in h. 264 decoders. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[20] C. Zhang, Y. Li, H. Zhou, X. Zhang, Y. Zheng, X. Zhan, X. Xie, X. Luo, X. Li, Y. Liu, et al. Automata-guided control-flow-sensitive fuzz driver generation. 2023.

[21] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu. APICRAFT: Fuzz driver generation for closed-source SDK libraries. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[22] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021.

[23] M. Zhang, C. Zhou, J. Liu, M. Wang, J. Liang, J. Zhu, and Y. Jiang. Daisy: Effective fuzz driver synthesis with object usage sequence analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023.