Bug detection in embedded environments by fuzzing and symbolic execution

Juraj Vijtiuk
Sartura,
Zagreb, Croatia
juraj.vijtiuk@sartura.hr

Luka Perkov
Sartura,
Zagreb, Croatia
luka.perkov@sartura.hr

Antonio Krog
Sartura,
Zagreb, Croatia
antonio.krog@sartura.hr

***Abstract* — OpenWrt is an Open Source GNU/Linux distribution designed for embedded devices which, although primarily targeting home routers, can run on residential gateways, other IoT (Internet of Things) devices. RIOT is an Open Source, real-time multi-threading operating system running on numerous devices that are typically found in IoT. Fuzzing is a software testing process that uses random inputs to track unusual behaviors and crashes. Symbolic execution is a software testing technique which executes the program with symbolic instead of concrete variable values to analyze which values cause what path of the program to execute. This paper offers an overview of how we used fuzzing and symbolic execution to automatically detect crashes within OpenWrt and RIOT projects. Special focus will then be placed on detected vulnerabilities and methods for identifying such issues. Results collected by subjecting two widely used embedded distributions to fuzzing tests will be used to draw conclusions on the state of security in Open Source embedded software.**

Keywords — security, fuzzing, OpenWrt, RIOT

I. INTRODUCTION

Embedded systems continue to drive technological advancements in many domains such as electronics, healthcare, communication, home appliances and many more. The constraints of embedded devices and the incorporation of network connectivity into these devices open numerous fields of vulnerabilities that are exploited by attackers. Widely published attacks targeting commercial routers, IoT deployments and other embedded systems [1, 2, 3] reiterated the importance of embedded systems meeting high security and privacy demands and minimizing the attack surface.

The state of security in embedded systems stems from the market and the practice of how they are produced. Chipset manufacturers race against time to release newer chipset revisions and stack Open Source and proprietary features on them to stand out from the competition. These chips are purchased by original device manufacturers (ODMs) who add their own set of features on them, verify that the board is functional and ship them further on. The chipset manufacturer then focuses on shipping the next version of the chip, and the ODM on upgrading its product to work with the newer chip. Even when the hardware is brand new, the software that ships with them is usually old, and no link in the production chain has an incentive to maintain older chips and products.

The result of this state are millions of deployed devices running unpatched, unmaintained software, and the vulnerabilities they expose are being increasingly exploited by attackers. A 2019 survey by the Cyber Independent Testing Lab (CITL) [4] examined more than 6,000 firmware images spanning more than a decade and found that firmware security and security standards have not significantly improved over the last fifteen years, and even the most recent firmware images show failures to implement basic security features.

Open Source software is widely regarded as a more secure model than proprietary software, in large part because it leverages large communities that regularly test the software and provide insights and fixes for security vulnerabilities. Despite this, most Open Source software projects are still afflicted by ineffective security practices. The findings in this paper show how Open Source software can benefit by employing even the most basic fuzz testing setups to pinpoint potential security weaknesses.

The remainder of this article is organised as follows. In section II, we describe the automatic bug detection methods that were used. Section III describes the testing setup and Section IV the results of the testing. Finally, Section V concludes this work, and Section VI lists areas of improvement for future work.

II. AUTOMATIC VULNERABILITY DETECTION METHODS USED

This section introduces two automatic vulnerability detection techniques used in this paper: fuzzing and symbolic execution.

A. Fuzzing

One of the most prominent available methods of automatic discovery of software security vulnerabilities is fuzzing. Fuzzing, also known as fuzz testing, is a software testing process that uses random inputs with the goal of finding unusual behaviors and crashes. The number of academic research and projects related to fuzzing is constantly on the rise, with GitHub alone hosting around three thousand public repositories as of this writing [5].

As most modern fuzzers are mutational, the procedure of the majority of fuzzers is the following [6]:

1. The fuzzer chooses a corpus of "seed" inputs that will be used to test the target program.

2. The fuzzer repeatedly produces new inputs from at least one seed and any observation about the program.

TABLE I. OVERVIEW OF USED FUZZERS

	AFL [8]	AFLplusplus [9]	honggfuzz [10]	gramfuzz [11]	Dharma [12]	Radamsa [13]	libFuzzer [14]	Angora [15]
Collecting observations	gray-box	gray-box	gray-box	black-box	black-box	black-box	gray-box	gray-box
Open Sourced	yes	yes	yes	yes	yes	yes	yes	yes
In-memory Fuzzing	yes	yes	yes				yes	yes
Program Analysis	yes	yes	yes				yes	yes
Mutation	yes	yes	yes			yes	yes	yes
Model-based				yes	yes	yes		

3. Every new seed is evaluated on the program to produce an on observation.

4. The fuzzer records any mutated inputs that produce "interesting" behavior.

5. The fuzzer stops when it times out or reaches a pre-defined goal such as finding a particular bug.

Depending on the degree of recorded observations, fuzzers are generally classified into three groups: black-box fuzzers that observe only the input and output behavior of the tested program; white-box fuzzers that generate tests by analyzing and exploiting the program's internals (e.g. source or binary code) and the information collected during test runs; and grey-box fuzzers that obtain only some information about the program's internals to avoid the overhead created by gathering additional observations and thus enable faster test runs.

Based on how input is generated, fuzzers are traditionally divided into generation-based (model-based) or mutation-based fuzzers [7]. Generation-based fuzzers generate test cases based on provided models (e.g. network protocols, formal grammars, file formats or languages) which describe the input that the program may accept. Mutation-based fuzzers, on the other hand, rely on using and modifying seeds to provide inputs whose type is supported by the program.

B. Symbolic execution

Symbolic execution is another common technique used to automatically detect issues in software. Although widely used, symbolic execution is somewhat less popular than fuzzing. Disadvantages of symbolic execution compared to fuzzing are a lack of understanding of symbolic execution, susceptibility to path explosion in larger targets, and difficulty of setup with real world projects.

Essentially, symbolic execution is performed by a symbolic execution engine that interprets the program and - instead of concrete values - assumes symbolic values for the program's variables. The key point lies in the ability to set constraints on those symbolic variables, which are then checked during symbolic execution. An example constraint could be to check whether a variable which should only have positive integer values has taken on a negative value. A survey of symbolic execution techniques is available at [35].

As described in the original KLEE paper [36], an advantage of symbolic execution is that it can easily generate input test cases for all the paths that have been found in the target, which can then be used to extend existing unit tests.

III. TESTING SETUP

The methodology used throughout this paper aims to follow best practices for evaluating fuzzers detailed by Klees et al. [6]. This section describes the setup used for our experiment.

A. Fuzzers

Table 1. presents a summary of the fuzzers utilized in this paper and the differences in the techniques they use. The table's concept is loosely based on classification provided by Manès et al. [7]. The first row indicates the fuzzer's classification based on the degree of recorded observations. The second row denotes whether the fuzzer is publicly available. The third row shows whether the fuzzer supports in-memory fuzzing, a process where the fuzzer takes a memory snapshot of the program under test and restores it on each fuzz iteration to avoid execution overhead. The fourth row shows whether the fuzzer performs static or dynamic analysis to gather information about the program's internals. The fifth row indicates if the fuzzer mutates the input to generate new test cases, while the sixth row indicates whether the fuzzer generates test cases based on a provided model. We opted to use LibFuzzer as a starting point for most targets because it is easier to set up compared to AFL and enabled us to test a wider array of targets more quickly. For targets that were already more thoroughly tested, like json-c, we employed a variety of tools with a higher setup cost such as AFL, honggfuzz and grammar-based fuzzers, and gathered a larger starting input corpus.

B. Symbolic execution

We opted to use the KLEE symbolic execution tool [16], in large part due to its suitability for working with projects where the source code is available, as opposed to

other popular symbolic execution engines aimed at binary symbolic execution (angr [17], Manticore [18]).

of the test trials is given further in this section. The last column of the table represents the number of cores the

TABLE II. TEST RESULTS

Program	Fuzzer(s) used	Trials	Timeout	Crashes	Number of cores/threads/fuzzer instances
ubus	AFL	1	12H	0	4/8/8
	Radamsa	1	24H	0	1/1/1
UCI	AFL	1	3H	5 (1 unique)	4/8/8
		2	1H	3 (1 unique)	4/8/8
		3	12H	0	4/8/8
	AFL + gramfuzz + Dharma	1	12H	0	4/8/8
	Radamsa	1	24H	0	1/1/1
libubox (base64 encoder)	LibFuzzer	1	12H	0	1/1/1
libubox (base64 decoder)	LibFuzzer	1	12H	0	1/1/1
libubox (blobmsg_parse)	LibFuzzer (+ASAN + UBSAN)	1	12H	0	1/1/1
libubox (blobmsg_parse_array)	LibFuzzer (MSAN)	1	12H	1	1/1/1
libjson-c (json_tokener_parse_ex)	AFL	1	24H	0	4/8/8
	AFLplusplus	1	12H	0	4/8/8
	honggfuzz	1	24H	0	4/4/1
	LibFuzzer	1	12H	0	1/1/1
	Angora	1	12H	0	4/4/1
rpcd (rpc_canonicalize_path)	LibFuzzer	1	12H	0	1/1/1
odhcpd (odhcpd_valid_hostname)	LibFuzzer (ASAN + UBSAN)	1	12H	0	1/1/1
	LibFuzzer (MSAN + UBSAN)	1	12H	0	1/1/1
RIOT (base64_encode)	LibFuzzer (ASAN + UBSAN)	1	12H	0	1/1/1
RIOT (base64_decode)	LibFuzzer (ASAN + UBSAN)	1	12H	0	1/1/1
RIOT (golay2412_decode)	LibFuzzer (ASAN + UBSAN)	1	12H	0	1/1/1
RIOT (golay2412_encode)	LibFuzzer (ASAN + UBSAN)	1	12H	0	1/1/1
RIOT (sys/checksum)	LibFuzzer (ASAN + UBSAN)	1	12H[a]	0	1/1/1
RIOT (sys/bitfield)	LibFuzzer (ASAN + UBSAN)	1	12H	0	1/1/1

a.For each checksum function

C. *Programs*

The following software components were submitted to testing: UCI, ubus, libubox, libjson-c, rpcd, odhcpd and RIOT.

D. *Performance measure*

Our experiments measured the number of crashes reported by the fuzzer over a period of time, with the same starting corpus. Remarks will be made with instances where crashes are caused by the same bug.

E. *Platform and configuration*

All fuzzing experiments were conducted on a single machine equipped with a four core Intel i7-8565U CPU with sixteen gigabytes of RAM, running Arch Linux. All fuzzers were built from source using the master branch, and all the target programs were also built from source using the master branch. For KLEE, we used the official KLEE Docker image and built the programs that were tested locally in that Docker image.

IV. RESULTS

Table 2. shows an overview of the fuzzing experiments and collected results. A detailed description

fuzzing jobs were running on, the number of threads used by the jobs (the CPU used had hyperthreading which showed up as eight cores), and the number of fuzzer instances that were run. The differences are most apparent between AFL and hongfuzz: honggfuzz automatically starts a parallel workload, resulting in one honggfuzz instance running four fuzz workers, one for each physical core. On the other hand, AFL's *afl-gotcpu* tool reported that eight cores were available, so we ran eight instances of AFL with each instance using two hyperthreads on four physical cores. All programs fuzzed with AFL were fuzzed with eight instances of AFL, namely one master instance and seven secondary instances.

A. *ubusd*

Initial testing was done on *ubusd*, an integral part of the OpenWrt's micro bus architecture named *ubus* [19]. The ubusd daemon provides an interface used by other daemons to register themselves. To fuzz ubusd, we wrote a custom harness that connected to the ubus UNIX socket and passed in files that were generated by the fuzzer. The harness starts by reading the contents of a file specified by the fuzzer into a buffer, then spawns a thread which reads the contents of the buffer and sends them to the ubus UNIX socket. The main thread continues with the start up

of the ubus server unmodified. We based this on existing approaches for fuzzing network servers [20]. The component was fuzzed with AFL for twelve hours using a single starting input file: a simple text file containing the text "list". The input is then randomly mutated with code coverage increases used as a measure of success by AFL. At first glance, the minimal input used here might seem overly simple to produce significant results. However, when testing smaller projects that do not use fuzzing, we discovered that using a single simple input as a starting point can quickly produce crashes in the majority of tested projects. In cases where no crashes are detected, a larger corpus of input files can be used for further processing and test trials. This trial produced no errors.

B. UCI

Next, we fuzzed OpenWrt's main configuration interface named *Unified Configuration Interface (UCI)*. A custom harness was written which imported configuration files generated by the fuzzer. The initial corpus contained a collection of OpenWrt UCI configuration files that were minimized using AFL's tools for minimizing test cases (*afl-tmin*) and corpus (*afl-cmin*) [21, 22]. On the first trial lasting three hours, AFL reported five crashes after two hours of fuzzing, all of which were caused by the same bug [23]. The second fuzzing run lasting an hour also revealed another bug in the first fifteen minutes, where numerous crashes reported as "unique" by AFL later turned out to be caused by the same bug [24]. A third run lasting twelve hours revealed no additional bugs.

To combat the lack of discovered vulnerabilities, we generated 1 million UCI configuration files by writing a UCI configuration grammar for the gramfuzz fuzzer. Even though none of the 5 million generated files caused crashes in UCI, they were leveraged in subsequent fuzzing iterations as a starting AFL corpus. The combined corpus of 1 million grammar-generated configurations and the previously used OpenWrt configurations were additionally minimized into a corpus containing eighty-nine input configs. An additional 1 million configuration files were generated using the grammar-based fuzzer Dharma. The grammar used for Dharma was based on the grammar used with gramfuzz.

Parsing the grammars generated by Dharma did not result in any new crashes, however it did result in additional input files which revealed new paths in UCI. The total number of UCI config input files after minimization was 149. UCI and ubusd were fuzzed using only vanilla AFL. Additionally, both the UCI and UBUS harnesses were fuzzed with Radamsa for twenty-four hours which found no issues.

C. libubox.

After UCI we moved onto fuzzing the *libubox* [25] OpenWrt library, which is used extensively throughout the OpenWrt project. We started fuzzing libubox using LLVM's LibFuzzer [14], which allows feeding fuzzed inputs to the library under test via a specific entrypoint. Initial fuzzing trials were done on libubox base64 encoder and decoder, but no bugs were found after twelve hours of fuzzing both components. We combined LibFuzzer with both the *AddressSanitizer (ASAN)* [26] and *UndefinedBehaviorSanitizer (UBSAN)* [27] and fuzzed *blobmsg_parse* for twelve hours. As we expected, this approach did not discover any new crashes since the OpenWrt community started fuzzing the same functions with LibFuzzer a few weeks earlier. Additionally, we combined LibFuzzer with the *MemorySanitizer (MSAN)* [28] and fuzzed libubox. Using this combination, we uncovered and fixed a series of issues with improper usage of flexible arrays in C structures in libubox, which appeared at multiple places in *blobmsg_parse* and *blobmsg_parse_array*. We also employed KLEE to test libubox. KLEE immediately found the out-of-bound read issues that were found by fuzzing with LibFuzzer and MSAN. With contributions by OpenWrt developers, these fixes were moved to generic blob functions and macros to prevent similar issues in other use cases utilizing these structures and to minimize future issues. We continued by testing the base64 encoder and decoder, *blobmsg_parse*, *blobmsg_parse_array*, and functions in *blobmsg_json.c* with KLEE, with no issues being found. Around the same time we were fuzzing *blobmsg_json* functions, a security advisory was released regarding a potential stack buffer overflow during serialization of JSON data in *blobmsg_format_json* [29]. The advisory did not describe how the issue was found, however the example proof of concept was triggered via ubus and rpcd, so we assume that deeper fuzzing of ubus or rpcd with AFL could - given enough time - find the issue, even though LibFuzzer did not find it in our case.

The analysis of the issue was done with a combination of manual code analysis, analysis in the rr debugger [30], and valgrind output for the program. We started by writing a simple main in *blobmsg_parse_array.c* which opened a file, read it into a buffer and passed it to the *blobmsg_parse_array* function. The target was then compiled without any sanitizers and ran through valgrind. The valgrind output is shown in Figure 1. The input file that caused the initial issue is shown in Figure 2. As the file contains binary data, the output of the xxd command which outputs the file's bytes in hexadecimal is shown.

```
==10829== Invalid read of size 2
==10829== at 0x109DFC: blobmsg_namelen
(blobmsg.h:74)
==10829== by 0x109446: blobmsg_check_name
(blobmsg.c:42)
==10829== by 0x1092DD: blobmsg_check_attr_len
(blobmsg.c:79)
==10829== by 0x109A63: blobmsg_parse_array
(blobmsg.c:159)
==10829== by 0x10A7BA: main (blobmsg.c:412)
==10829== Address 0x4a2e2b4 is 0 bytes after a
block of size 4 alloc'd
==10829== at 0x483877F: malloc
(vg_replace_malloc.c:309)
==10829== by 0x10A773: main (blobmsg.c:408)
```

Figure 1. Valgrind output

```
$ xxd crash-
a3585b70f1c7ffbdec10f6dadc964336118485c4

00000000: 0300 0004
```

Figure 2. xxd output

The *blobmsg_parse_array* function tries to interpret the received data as a *blob_attr* structure, which consists of a *uint32_t* followed by a flexible array member. The existing check in libubox checked whether the received data is smaller than *sizeof(struct blob_attr)*, however as the flexible array member's size is counted as 0, the input shown above passes that check since it is exactly 4 bytes large. The issue then appears when accessing the flexible array member, even though it is empty in this case. After adding checks to prevent this issue, we continued fuzzing *blobmsg_parse_array*, which revealed additional issues of the same type. The *blob_attr* flexible array member is parsed into another structure that ends with a flexible array member *blobmsg_hdr*. The same out-of-bounds read issue appeared, so similar checks were added to prevent that issue. The same type of issues were noticed and fixed in *blobmsg_parse*. However, as the two above mentioned functions are not the only places where *blobmsg_hdr* and *blob_attr* are used, OpenWrt developers expanded the fix to account for other places where the issue might appear.

D. json-c

We fuzzed *json-c* for twenty-four hours using LibFuzzer compiled with ASAN and UBSAN and utilized three LLVM fuzz jobs, each one running on a separate core, but did not discover any issues. Next, we generated 1 million JSON files with Dharma, but this approach also did not find any issues.

Afterwards, we fuzzed json-c using AFL and leveraged a corpus containing JSON files collected from the Internet and JSON files generated by Dharma. This approach did not discover any issues in twenty-four hours. The same corpus was utilized when fuzzing json-c with hongfuzz for twenty-four hours, but this approach also did not find any issues. Fuzzing json-c with AFLplusplus for twelve hours also discovered no issues, although AFLplusplus proved to be much more successful at discovering paths than AFL: AFL discovered around 600 paths in twenty-four hours, while AFLplusplus discovered around 1300 paths in twelve hours with the same corpus. The AFLplusplus configuration consisted of eight instances: one master instance running with the exploit power schedule, three secondary instances running with a COE power schedule, two secondary instances running with the fast power schedule and two secondary instances running with the explore power schedule. Since we were unable to find bugs using fuzzing, we tested json-c with KLEE by writing and running a simple wrapper that calls *json_tokener_parse_ex* with a symbolic JSON string, but this approach also did not find any issues. Ultimately, no bugs were found in json-c, which was expected, as json-c is regularly fuzzed by Google's distributed fuzzing project, OSS-Fuzz [31].

E. RIOT

When we started looking into what part of RIOT would be best to test with fuzzers, we first consulted previous work in this field. A number of GitHub issues and pull requests already exist on the RIOT-OS github page [32]. Previous issues with coap parsing and the TCP stack were found and fixed by fuzzing. A custom black-box fuzzer for fuzzing CoAP server side implementations *fuzzcoap* was also used to test RIOT's CoAP code [33]. The RIOT 2017 Summit featured a presentation of work by Eric Sesterhenn, which resulted in 40 issues being reported in RIOT and in projects related to RIOT. The work also used fuzzing procedures and utilized honggfuzz, LibFuzzer, radamsa and AFL fuzzers.

Initially, we started to fuzz RIOT with AFL by writing a custom RIOT-OS application which would take the role of the harness and pass AFL inputs to the GNRC networking functions available in RIOT. However, due to time constraints we did not manage to successfully build such a harness, as RIOT abstracts away a lot of the build process behind Makefiles, so we ran into various issues while trying to build the custom application and instrumenting the core of RIOT. Instead, due to the ease of setup, we opted to fuzz as many separate functions with LibFuzzer. Other contributors have managed to set up fuzzing with AFL, and as of this writing a pull request was opened on RIOT-OS's GitHub with support for fuzzing the GNRC networking functions with AFL [34]. The author of the pull request has previously managed to find a dozen crashes and security issues in the fuzzed targets by using the setup available in the pull request.

Similarly to libubox, we started by fuzzing the base64 encoder and decoder for six hours with no starting inputs and instrumented with ASAN and UBSAN, but this approach did not result in any detected crashes. We continued by fuzzing the *golay2412_decode* and *encode* elliptic curve functions with the same configuration, but did not find any issues. Next, we fuzzed RIOT-OS's bitfield implementation and the checksum functions using the same configuration, but this did not reveal any bugs.

V. CONCLUSION

This paper, the experiments conducted within it and the collected results are not meant to provide a comprehensive analysis of available automatic bug detection techniques. Instead, the insights from this paper aim to emphasize the benefits that even widely used, reliable Open Source software can gain from elementary, easy-to-setup fuzzing tests. The wide list of available fuzzers can be used and combined to discover shallow bugs within the first few hours of testing. The fact that most modern software is fuzzed for weeks if not months at a time, and that all of the programs fuzzed in this paper were fuzzed for at most a day and discovered crashes in the process emphasizes the need to include fuzzing as another form of testing Open Source projects. The differences are especially noticeable when comparing json-c, a thoroughly tested library, for which we did not manage to find any issues with multiple approaches, with UCI and libubox which have not been thoroughly tested yet. Fuzzing with LibFuzzer resulted mostly in shallow, easily-reachable crashes, while a longer-lasting fuzzing run using a starting corpus of inputs with decent coverage should result in discovering more obscure bugs.

In addition to fuzzers, this paper highlighted the benefits from another underutilized form of automated bug detection: symbolic execution. Although easier to set up elementary testing, symbolic execution quickly revealed bugs that might require more thorough fuzzing

setups to detect as well as pinpointed issues that go undetected by the majority of utilized fuzzers. Symbolic execution tools, especially KLEE, provide an added benefit where even in cases where no bugs are discovered, the tools generate inputs for all unique paths discovered during exploration. These inputs can later on be used to expand existing or to create new unit tests that improve the security and stability state of tested programs. In-depth testing with KLEE, however, still requires the tester to be familiar with the target codebase.

VI. Future work

The grammars generated with gramfuzz and Dharma were mostly invalid. We will invest the effort to write better grammars which will cover a wider array of types of invalid UCI configurations, as well as generate more valid configurations.

In this paper we only managed to fuzz RIOT-OS with LibFuzzer by separately fuzzing functions of interest. Future work could focus on fuzzing RIOT-OS with AFL by using a custom RIOT-OS application as the harness. This would enable discovering deeper bugs in the code, since LibFuzzer is more suited for discovering shallower bugs. Another approach would be to reuse the test cases generated by KLEE as additions or starting points for a fuzzing corpus, which would improve fuzzer coverage.

Although KLEE has made huge advancements by providing its own symbolic versions of libc and the POSIX environment, we encountered issues when testing software that used GNU extensions to libc, or GNU-specific functions. The workaround in this case was to manually patch the software to replace the functionality with non-GNU functions. Another area of future work would consist of adding additional implementations of such functionality.

References

[1] S. Larson, “Massive cyberattack targeting 99 countries causes sweeping havoc”, http://money.cnn.com/2017/05/12/technology/ransomware-attack-nsa-microsoft/index.html, 13. 5. 2017

[2] C. Cimpanu, “Brazil is at the forefront of a new type of router attack”, https://www.zdnet.com/article/brazil-is-at-the-forefront-of-a-new-type-of-router-attack/, 12. 7. 2019

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas and Y. Zhou, “Understanding the Mirai Botnet”, In USENIX Security Symposium, 2018.

[4] P. Roberts, “Huge Survey of Firmware Finds No Security Gains in 15 Year”, https://securityledger.com/2019/08/huge-survey-of-firmware-finds-no-security-gains-in-15-years/, 14. 8. 2019.

[5] GitHub, “Public Fuzzers”, https://github.com/searchq=fuzzing&type=Repositories, 25. 1. 2020.

[6] G. Klees, A. Ruef, B. Cooper, S. Wei and M. Hicks, “Evaluating Fuzz Testing”, In 2018 ACM SIGSAC Conference on Computer and Communications, 2018.

[7] V. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz and M. Woo, “The Art, Science and Engineering of Fuzzing: A survey”, In IEEE Transactions on Software Engineering, 2018.

[8] M Zalewski, “American Fuzzing Lop (AFL)”, http://lcamtuf.coredump.cx/afl/, 26. 9. 2019.

[9] M. Hauser, H. Eißfeldt, A. Fioraldi, D. Maier, “American Fuzzing Lop Plus Plus (AFL++)”, https://github.com/vanhauser-thc/AFLplusplus, 31. 12. 2019.

[10] R. Swiecki and F. Gröbert, “honggfuzz”, https://github.com/google/honggfuzz, 7. 12. 2019.

[11] J. Johnson, “gramfuzz”, https://github.com/d0c-s4vage/gramfuzz, 2. 1. 2020.

[12] Mozilla Security, “Dharma”, https://github.com/MozillaSecurity/dharma, 5. 12. 2019.

[13] A. Helin, “Radamsa”, https://gitlab.com/akihe/radamsa,7. 5. 2019.

[14] The LLVM Project, “LibFuzzer”, https://llvm.org/docs/LibFuzzer.html, 13. 1. 2020.

[15] P. Chen, H. Chen, “Angora”, https://github.com/AngoraFuzzer/Angora, 17. 7. 2019.

[16] The KLEE Team, “KLEE LLVM Execution Engine”, https://klee.github.io/, 2019.

[17] Shoshitaishvili, Yan and Wang, Ruoyu and Salls, Christopher and Stephens, Nick and Polino, Mario and Dutcher, Audrey and Grosen, John and Feng, Siji and Hauser, Christophe and Kruegel, Christopher and Vigna, Giovanni, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”, In *IEEE Symposium on Security and Privacy*, 2016

[18] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, “Manticore”, https://github.com/trailofbits/manticore, 12. 11. 2019.

[19] The OpenWrt project, “ubus (OpenWrt micro bus architecture)”, https://openwrt.org/docs/techref/ubus, 11. 1. 2020.

[20] javi, “Fuzzing Apache httpd server with American Fuzzy Lop + persistent mode” , https://animal0day.blogspot.com/2017/05/fuzzing-apache-httpd-server-with.html, 7. 5. 2017.

[21] D. Stender, “afl-tmin(1)”, http://www.tin.org/bin/man.cgi?section=1&topic=afl-tmin, 20. 1. 2020.

[22] D. Stender, “afl-cmin(1)”, http://www.tin.org/bin/man.cgi?section=1&topic=afl-cmin, 20. 1. 2020.

[23] Luka Kožnjak, Juraj Vijtiuk, openwrt-devel mailing list, “[OpenWrt-Devel] [PATCH] file: fix segfault in uci_parse_option”, http://lists.openwrt.org/pipermail/openwrt-devel/2019-December/026597.html, 28. 12. 2019.

[24] Luka Kožnjak, Juraj Vijtiuk, openwrt-devel mailing list, “[OpenWrt-Devel] [PATCH] file: fix segfault in uci_parse_config”, http://lists.openwrt.org/pipermail/openwrt-devel/2019-December/026600.html, 28. 12. 2019.

[25] The OpenWrt Project, “libubox”, https://git.openwrt.org/project/libubox.git, 20. 1. 2020.

[26] The LLVM Project, “Clang 11 documentation - AddressSanitizer”, http://clang.llvm.org/docs/AddressSanitizer.html, 2020.

[27] The LLVM Project, “Clang 11 documentation - UndefinedBehaviorSanitizer”, http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2020.

[28] The LLVM Project, “Clang 11 documentation - MemorySanitizer”, http://clang.llvm.org/docs/MemorySanitizer.html, 2020.

[29] P. Štetiar, J. Wich, “Security Advisory 2020-01-22-2 - libubox tagged binary data JSON serialization vulnerability (CVE-2020-7248)”, https://openwrt.org/advisory/2020-01-31-2, 31. 1. 2020.

[30] Mozilla, “rr project homepage”, https://rr-project.org/, 2017.

[31] Google, “OSS-Fuzz”, https://github.com/google/oss-fuzz, 2020.

[32] “RIOT - The friendly OS for IoT”, https://github.com/RIOT-OS/RIOT, 31. 10. 2019.

[33] B. Melo, “FuzzCoAP”, https://github.com/bsmelo/fuzzcoap, 23. 6. 2018.

[34] “RIOT - The friendly OS for IoT”, “Add AFL-based fuzzing setup for network modules #13157”, https://github.com/RIOT-OS/RIOT/pull/13157, 17. 1. 2020.

[35] E. J. Schwartz, T. Avgerinos, D. Brumley, “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”, In IEEE Symposium on Security and Privacy, 2010.

[36] C. Cadar, D. Dunbar, D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”, 2008.