So Many Fuzzers, So Little Time*

Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack

Clément Poncelet
clement.poncelet@it.uu.se
Uppsala University
Uppsala, Sweden

Konstantinos Sagonas
kostis@it.uu.se
Uppsala University
Uppsala, Sweden
National Technical University of Athens
Athens, Greece

Nicolas Tsiftes
nicolas.tsiftes@ri.se
RISE Research Institutes of Sweden
Stockholm, Sweden
KTH Digital Futures
Stockholm, Sweden

ABSTRACT

Fuzz testing ("fuzzing") is a widely-used and effective dynamic technique to discover crashes and security vulnerabilities in software, supported by numerous tools, which keep improving in terms of their detection capabilities and speed of execution. In this paper, we report our findings from using state-of-the-art mutation-based and hybrid fuzzers (AFL, Angora, Honggfuzz, Intriguer, MOpt-AFL, QSym, and SymCC) on a non-trivial code base, that of Contiki-NG, to expose and fix serious vulnerabilities in various layers of its network stack, during a period of more than three years. As a by-product, we provide a Git-based platform which allowed us to create and apply a new, quite challenging, open-source bug suite for evaluating fuzzers on real-world software vulnerabilities. Using this bug suite, we present an impartial and extensive evaluation of the effectiveness of these fuzzers, and measure the impact that sanitizers have on it. Finally, we offer our experiences and opinions on how fuzzing tools should be used and evaluated in the future.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software defect analysis**.

KEYWORDS

Software security, security testing, fuzz testing, coverage-guided fuzzing, hybrid fuzzing, IoT, Contiki-NG

ACM Reference Format:
Clément Poncelet, Konstantinos Sagonas, and Nicolas Tsiftes. 2022. So Many Fuzzers, So Little Time: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556946

1 INTRODUCTION

A famous quote attributed to Dijkstra states that *"If debugging is the process of removing software bugs, then programming must be the process of putting them in."* Perhaps due to most programming still done by humans, all non-trivial software contains bugs. Some of these bugs are serious vulnerabilities and often exploitable. Moreover, there exist application domains where pretty much *all* serious software vulnerabilities are exploitable. One such domain, of interest to us, is that of IoT systems; more specifically, that of operating systems for resource-constrained IoT devices. In this domain, a software bug that crashes the OS has severe consequences: it results in DoS at best or, worse, brings down the whole IoT system (e.g., a home alarm, an environmental monitoring system, etc.) permanently. So, naturally, developers and researchers interested in such OSes want techniques and tools that help them find such software bugs and fix them before they make it into OS releases.

In recent years, a successful technique to discover crashes in software is *fuzz testing* [16] or simply *fuzzing*. In this paper, we specifically focus on *coverage-guided grey-box fuzzing* [27, 30] techniques. Since mid 2018, we have jumped on the so-called *fuzzing hype-train* [19], and have been using various mutation-based grey-box and hybrid fuzzers to discover bugs in Contiki-NG [17], the Next Generation OS for IoT devices forked from its predecessor, Contiki. During the same period, we have been following closely the research published in the area, esp. that related to hybrid fuzzing techniques, and trying out fuzzers which were already available or ever since released as open source. Some research questions that naturally come to mind from engaging in such a project are:

RQ.1 (Effectiveness) Are hybrid fuzzers superior in exposing bugs and vulnerabilities than mutation-based fuzzers?

RQ.2 (Efficiency) Do some of the fuzzers employ techniques which allow them to expose bugs fast? If so, which?

RQ.3 (Consistency) Are any fuzzer implementations able to expose (some of) the bugs in all/most of their runs?

This paper provides its answers to these questions by following a systematic evaluation. More precisely, we employ a total of eight different fuzzers, four *mutation-based* (two AFL variants [30], Honggfuzz [27], and MOpt-AFL [14]) and four recent *hybrid* fuzzers (Angora [4], QSym [29], Intriguer [5], and SymCC [21]) to produce inputs that reliably trigger crashes in the code base of Contiki-NG's low-power IPv6 stack. Over a period of a bit more than three years, we have been precisely documenting these bugs, reporting them to Contiki-NG's developers so that they get fixed, and obtaining CVEs for them when the pull requests which fixed them got merged into its development branch. Our efforts have made Contiki-NG more robust and secure, but have also made us more knowledgeable. Based on the experience we have gained, we have formed some opinions both on how fuzzing tools should be used and on how they

*With apologies to the title of Miquel Brown's song from 1983.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556946

should be evaluated and compared to each other. Of course, we are not the first to notice and report views and findings on these topics. For example, several papers published in top conferences and journals [11–13] report that various questionable practices have been used to evaluate and compare fuzzing techniques and tools, and that there is a clear need for better, more "real-world", and more challenging benchmarks. In this paper, besides corroborating experiences and recommendations on these topics published in the literature, we contribute some of our own. We also mention when we do not fully agree with some of the above papers, and where our findings differ from them and why.

We hold that reporting such user experiences is valuable for the fuzzing research community anyway, but we go beyond that. We have created benchmarking infrastructure that can be used to evaluate existing and, more importantly, future fuzzing techniques and tools on the Contiki-NG code base. Our paper's artifact, which is available on GitHub, currently comes with a suite of a total of eighteen *ground truth* bugs, all discovered during our fuzzing train journey, that correspond to vulnerabilities at different layers of Contiki-NG's code base, and are increasingly difficult to expose. The scripts that the artifact contains use the Git commit history in order to rewind and replay the evolution of Contiki-NG's code base starting from a particular commit where a number of these eighteen vulnerabilities have been fixed (and the remaining have not). We also evaluate the eight fuzzers mentioned above on this suite, and report our findings from running several 24-hour experiments for individual ground truth bugs. As we will see, some of the bugs in our suite are currently quite challenging for most contemporary mutation-based and hybrid fuzzers. This is due to inherent characteristics of Contiki-NG's code base: it implements a *network stack*, and in order to reach some particular layer of its code base, the input has to pass the checks of other layers.

Quite often, fuzzing is aided by *sanitizers*, i.e., runtime analysis tools that employ instrumentation to discover crashes or undefined behaviour in software. Sanitizer-aided fuzzing has pros and cons: Sanitizers help in exposing and triaging bugs and vulnerabilities more accurately [23], but they also impose a non-negligible runtime overhead, which means that fuzzers explore (significantly) fewer executions within a given time budget when using them. As far as we know, there is no published work that investigates this tradeoff, i.e., reports whether sanitizers have a clear positive effect on bug discovery or whether it is better to not slow down the fuzzer and allow it to explore more inputs during a time-limited fuzzing campaign. This is the fourth and last question we investigate.

RQ.4 (Sanitizer Impact) Do sanitizers pay off for their runtime overhead in terms of exposing more vulnerabilities within a time-limited fuzzing run?

In summary, this work:

- Reports findings from using eight state-of-the-art mutation-based and hybrid fuzzers on a non-trivial code base, that of the Contiki-NG OS, to discover and fix serious vulnerabilities, many of which have CVEs (cf. Table 2).
- Presents an impartial evaluation of the effectiveness of these eight fuzzers, not with an aim to declare a "winner" among them, but so as to highlight those that are expected to perform well on code bases with Contiki-NG's characteristics.
- Evaluates the impact that two sanitizers (AddressSanitizer and Effective Type Sanitizer) have on fuzzing tools.
- Offers a new, quite challenging, bug suite for evaluating fuzzers on real-world software bugs.
- Comes with a Git-based platform as artifact, and argues for the pros of using such a benchmarking approach for evaluating fuzzers in terms of their ability to expose vulnerabilities.

The remainder of this paper starts by presenting relevant information about the fuzzing tools we use and a brief review of Contiki-NG's layered architecture (§2). The next and main sections of this paper present the details of our ground truth suite with the results of our evaluation (§3) and our investigation on sanitizer impacts on fuzzing tool effectiveness (§4). The paper ends with related work (§5) and some final remarks.

2 BACKGROUND

First, in §2.1, we overview the fuzzing techniques and tools we use in our evaluation. Note that we do not aim to present an exhaustive account of fuzzing technology; instead, we concentrate on recently proposed and state-of-the-art mutation-based and hybrid fuzzers, focusing on characteristics that can influence answers to our research questions. We then briefly present Contiki-NG in §2.2.

2.1 Fuzzing

Fuzz testing [15, 16] discovers software bugs by randomly generating inputs and feeding them to a program under test (the "*target*"). Since most targets expect that their inputs have a specific structure, *mutation-based fuzzers* demand an initial set of inputs ("*seeds*") from their users, and start applying small mutations to them. Their aim is to generate mutants satisfying the target's consistency checks and manage to penetrate its code deeply. *Coverage-guided grey-box fuzzers*, such as the popular American Fuzzy Lop (AFL) [30], instrument the target lightly to return to the fuzzer some feedback after each target execution (e.g., which code blocks were visited, information which is maintained in a coverage bitmap in the case of AFL). AFL's instrumentation is implemented by two companion utilities that act as a drop in replacement for `gcc` and `clang`: `afl-gcc` and `afl-clang` use an *ad hoc* script to instrument the target at the assembly level. A third utility, `afl-clang-fast` [1], uses true compiler-level instrumentation, claiming to produce a target configuration that can be up to ten times faster than the corresponding target produced by `afl-clang`.[1] Mutation-based fuzzers implement a set of *mutators* (i.e., operators that modify the input in some way), and must choose how frequently to apply each of them. In essence, this is a priority scheduling (i.e., an optimization) problem. However, it is a dynamic optimization problem, because the mutator that will generate an interesting mutant depends on the current input, and may greatly vary during a fuzzing session. MOpt [14] proposes to use a *particle swarm optimization* algorithm to compute the optimal selection probability distribution over those mutators,

[1] In our evaluation, we include two variants of AFL: `afl-gcc` and `afl-clang-fast` (which we abbreviate AFL-cf). AFL-cf highlights the impact that instrumentation can have on AFL's effectiveness, and also on the effectiveness of hybrid fuzzers that also instrument using `afl-clang-fast`. As we will see, our experiences and results agree with observations made by Poeplau and Aurélien [20] about how Intermediate Representations or instrumentation for symbolic execution may affect not only fuzzers' speed but also their ability to detect bugs. More on that in §3.

instead of using a fixed mutator selection strategy. For our evaluation, we used MOpt-AFL, the authors' algorithm implementation within AFL (henceforth referred to as MOpt, for simplicity). We also include Honggfuzz [27] to our evaluation, a tool recently reported to outperform many other fuzzers in an experimental study similar to ours [2]. This mutation-based fuzzer relies on the ptrace API together with UndefinedSanitizer [28] to detect target's crashes. With this implementation, Honggfuzz provides a different capability of detecting vulnerabilities than AFL, and provides better information to the users about why a crash occurred.

Various researchers have proposed *hybrid fuzzers* [18], i.e., fuzzing tools that also run some heavier code analysis tool(s) alongside a mutation-based fuzzer. A hybrid fuzzer feeds its dynamic analysis component with fuzzer's mutants in order to generate inputs that a grey-box fuzzer is unlikely to generate due to its lack of knowledge about the program. The inputs produced by the dynamic analysis component are then fed back into the mutation-based fuzzer, which in turn can then use them to penetrate new 'easily-reachable' code. Driller [26] was the first hybrid fuzzer to popularize this idea, by running a concolic execution engine whenever AFL did not show any signs of progress (i.e., in periods when AFL failed to come up with mutants that increase code coverage). Driller's approach was subsequently adopted and improved upon by the authors of QSym [29]. QSym's concolic execution engine executes an input generated by mutation and, for every encountered branch, it tries to generate inputs that follow the alternative paths. QSym implements a complete concolic engine executing binary instructions, and adds heuristics to further speedup the execution, trading the strict soundness requirements of conventional concolic execution for better performance [29]. Interestingly, Angora [4], which is also a hybrid fuzzer, has opted for a different approach. It uses *taint analysis with a machine learning algorithm* to generate inputs leading to alternative paths. Taint analysis provides target's instructions and the input bytes they accessed during an execution. With that knowledge, Angora focuses on 'flipping' a target branch by modifying only the tainted bytes of the input. Additionally, Angora extends AFL's coverage feedback component to maintain information at finer granularity, which however requires more time and memory. Intriguer [5] is a fuzzer that combines taint analysis with *symbolic execution*. From a tainted trace, it removes any `mov`-like instructions, and reconstructs the symbolic expressions by mapping input to machine instructions' values. As a result, Intriguer's symbolic execution component becomes cheaper at the cost of the previous steps. Finally, SymCC [21], a more recent hybrid fuzzer, directly embeds the symbolic process of executing an input into the target. This way, for a given input, the instrumented target executes symbolically in native speed. SymCC's technique allows to use any symbolic reasoning tool as a backend, and proposes two implementations: one simply creating symbolic expressions and offloading them to an SMT solver; the other using the concolic execution engine of QSym. We have selected SymCC-QSym (SymCC with QSym backend) for our evaluation, because it is the faster implementation.

2.2 Contiki-NG Network Stack

The fuzzers' target is Contiki-NG [17], which is an open-source operating system designed for resource-constrained IoT devices. Its main feature is its low-power IPv6 stack, which includes a variety of

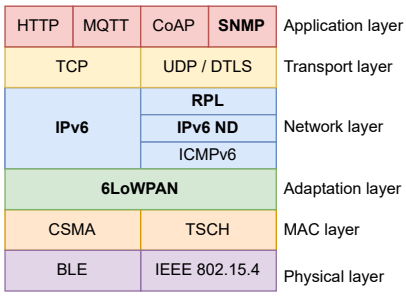

Figure 1: Protocols at different layers in the Contiki-NG low-power IPv6 stack. The components in bold show where our fuzzing experiments have revealed code vulnerabilities.

standard communication protocols implemented in a compact manner. Figure 1 shows the key components of Contiki-NG's network stack arranged at different layers.

At its lowest layers, where incoming packets enter the network stack in a deployed network, Contiki-NG supports communication standards such as IEEE 802.15.4 and Bluetooth Low Energy (BLE). For medium access control, Contiki-NG provides the option between a basic Carrier Sense Multiple Access (CSMA) protocol and the IEEE 802.15.4 Time-Slotted Channel Hopping (TSCH) protocol. Between the MAC layer and the network layer resides the 6LoWPAN adaptation layer, which performs header compression and fragmentation of IPv6 packets. At the network layer, the central component is the IPv6 implementation, which is based on the original μIP in the Contiki OS [8]. It relies on ICMPv6 for control messages, and IPv6 neighbor discovery (IPv6 ND) for translating IPv6 addresses to link-layer addresses and for keeping track of neighbors and routers. For routing within multi-hop wireless networks, Contiki-NG implements the Routing Protocol for Low-power and lossy networks (RPL), which uses ICMPv6 to exchange messages. At the transport layer, Contiki-NG supports both UDP and TCP communication. Both of these implementations are integrated deeply into the μIP component in order to achieve small code size. Furthermore, Contiki-NG supports the DTLS protocol to provide transport layer security on top of UDP. At the application layer, Contiki-NG supports the Constrained Application Protocol (CoAP). In addition, the implementation of the network management protocol SNMP relies on UDP. On top of TCP, one can use either MQTT or HTTP as application layer protocols.

Each of these protocol implementations can be a possible attack target for anyone who has the capability to inject packets into it.

One important point to keep in mind for our work is that, when fuzz testing the Contiki-NG network stack, one has to consider its layered architecture when deciding where to inject packets. One option is to inject all packets at the 6LoWPAN layer or lower, but this entails that a packet will have to pass a large number of checks of different headers before it reaches an upper layer. Hence, the fuzzers will have difficulty in achieving an adequate coverage in the protocol implementations at the upper layers of the network stack; e.g., in the CoAP implementation. Alternatively, one can inject packets directly into the protocol implementation of interest for more focused fuzz testing, at the expense of not covering key parts of the network stack. In our experiments, we focus on injecting packets at the IPv6 and 6LoWPAN layers, which are the lowest

points of the stack where significant packet processing occurs, and where input data has the potential to reach code in a variety of protocol implementations on top of these layers.

3 GROUND TRUTH EXPERIMENTS

In this section, we evaluate the eight fuzzers on the set of eighteen vulnerabilities that we have detected and fixed using fuzz testing. We run fuzzing campaigns on Contiki-NG and compare fuzzing tools' performance to answer the first three research questions.

Fuzzer Selection and Some Experiences. We have selected fuzzers to include in our evaluation as follows. In fall 2018, we started fuzzing Contiki-NG's μIP layer using AFL. We continued using AFL on μIP, with moderate success (we discovered the first three vulnerabilities of Table 2), until we hit a wall and no more bugs could be found with the test harnesses we were using at the time. We therefore turned our attention to hybrid fuzzers that, at least in principle, are more powerful, and tried out Angora and Driller. However, we experienced usability issues with Driller, and quickly abandoned it for QSym. Using Angora and QSym, we were able to trigger more bugs. Up to that point, all fuzzers were run on the laptops and servers of our group, and maintenance was painful whenever software on these machines was updated. In spring 2020, we created Docker containers for running the fuzzers, which allowed us to quickly include Intriguer in our tool set. In the fall of 2020, we learned about MOpt and SymCC, but it was easy to include them in our toolset, because it had evolved further at that point. This turned out to be a good choice, because we were able to discover one more, hard to trigger and reproduce, Contiki-NG vulnerability using them.

We remark at this point that the set of fuzzers that we use has a small common intersection with those used in three recent papers, which also propose benchmark platforms for evaluating fuzzers (Magma [11], UniFuzz [13], and one based on FuzzBench [2]). Moreover, these three papers and ours investigate different sets of research questions, but, even in the common ones, their conclusions slightly differ from ours. More on this in §5.

Platform and Configuration. Our artifact runs the fuzzers within Docker containers. The configurations these use are shown in Table 1. To run trials in parallel, 20 to 30 at a time, the machine we used for the measurements we report is a server with two Intel(R) Xeon(R) Platinum 8168 CPUs (2.70GHz with 24 physical cores each, i.e., a total of 48 physical cores / 96 with hyperthreading) and 192GB of RAM running Debian 10.7. For all the trials, two instances of AFL (MOpt-AFL for MOpt) are running using AFL parallelization mode. More precisely, every AFL or MOpt-AFL trial runs two processes, Honggfuzz runs with two threads in its thread pool, and every hybrid fuzzer runs three processes (two instances of AFL plus one more for the symbolic/concolic execution component).

For MOpt, we set MOpt-AFL's pacemaker mode option to `-L 0` for all trials. (We picked this setting, which controls the time after which AFL's deterministic mutations are disabled, because it gave the best performance after multiple 24-hour trials.) Due to bad detection of free CPUs from AFL within our configuration, we disabled AFL's CPU confinement. AFL's execution speed is around 1,200 execs/sec. We also used RAMDisk to avoid slow disk accesses.

Table 1: Docker image configurations for our experiments.

Tool	Version	Ubuntu OS	AFL Instrumentation	Compiler
AFL-gcc	2.57b	16.04 LTS	afl-gcc	gcc 5.4.0
AFL-cf	2.57b	16.04 LTS	afl-clang-fast	clang 3.8.0
MOpt	e3e6936	16.04 LTS	afl-gcc	gcc 5.4.0
Honggfuzz	0b4cd5b1	20.04 LTS		clang 8.0.1
Angora	1.2.2	16.04 LTS	afl-gcc	gcc 5.4.0
QSym	4fa4363	16.04 LTS	afl-gcc	gcc 5.4.0
Intriguer	4d41176*	16.04 LTS	afl-gcc	gcc 5.4.0
SymCC	e29fc5a	20.04 LTS	afl-clang	clang 10.0.0

* We have fixed a disk space issue for Intriguer.

Seeds. We use two entry points to inject fuzzed data packets at different layers in the Contiki-NG network stack: one to the μIP module and one to the 6LoWPAN module. As a result, we use two different sets of seeds. However, for a given entry point, all fuzzers and all trials use the *same* set of seeds. For μIP, the seeds are built from four well-formed IPv6 packets. They all contain a valid IPv6 header followed by different optional extension headers (Hop-by-hop, Routing, or ICMPv6) with possible options (such as RPL configuration). For 6LoWPAN, we follow the corresponding standard [22] and encapsulate IPv6 packets after 6LoWPAN's lower-layer headers. This provides a valid input for the 6LoWPAN entry point and still covers what the μIP seeds cover. Another possibility is to generate inputs with 6LoWPAN compression or fragmentation header. However, for this paper, we want to generate simple and valid seeds to evaluate how well fuzzing tools explore different headers of each protocol. Then, we use the AFL's corpus and test case minimization tools (afl-cmin and afl-tmin) to optimize the sets of seeds for AFL—removing redundant inputs and trimming the files—and use the resulting sets to initialize our fuzzers.

Benchmark Trials and Numbers Reported. All the times we present in the tables of this section are averages of a total of ten 24-hour trials for each fuzzer. Each fuzzer trial specifically targets a singleton set consisting of some particular ground-truth bug (e.g., `uIP-len`). After each 24-hour trial finishes, a post-processing script checks all "unique crashes" and hangs detected by the fuzzer, validates whether some of them correspond to *a bug in the target set* and, if so, extracts the earliest time that each such bug was exposed. All such times from ten trials are then averaged and presented in tables of this section as *mean time-to-exposure*. We use the 🕒 symbol to denote that a bug was not detected in any of the ten trials. In the cases where non-trivial bugs are exposed *consistently* (i.e., in all ten out of ten trials) by some fuzzers, and this happens fast or in time that is considerably shorter than that of most other fuzzers that also consistently expose these bugs, we highlight those times.

Design of "Ground Truth" Benchmark Suite. Our benchmark suite consists of selected vulnerabilities we discovered in the code base of Contiki-NG using fuzzing. The vulnerabilities are listed in Table 2; we invite the reader to read its detailed caption at this point.

Let us provide some additional information about these vulnerabilities and our experiences. We started with fuzzing Contiki-NG's code base, at its IPv6 network layer, with μIP as entry point using AFL 2.36. Even on the first day, AFL exposed the first crash (`uIP-overflow`) within the first half hour of fuzzing. However, after running for about two hours, AFL also reported around 100 more "unique crashes" which turned out to have the same culprit as the

Table 2: Vulnerabilities selected for our experiments. Each of them is given an identifier (Id), has a pull request that fixes it (PR♯), which in turn is associated with two Git commit SHAs before and after the fix was merged. The last four columns show the protocol implementation where the bug was located, the CVE number assigned to it (if any), a short description of the error, and the fuzzer that first discovered it, although this mostly reflects on when we first started using each fuzzer. Vulnerabilities are shown in chronological order with which their fixes were *merged*. This order is almost identical to the chronological order in which vulnerabilities were discovered, with two exceptions (the pairs `6LoWPAN-ext-hdr`, `ND6-overflow` and `SNMP-oob-varbinds`, `SNMP-validate-input`) for which the vulnerabilities were discovered in time close to each other, but in the opposite order.

Id	PR♯	Commit SHAs	Protocol	CVE	Error description	Discovered by
`uIP-overflow`	813	`a1cba56–ea6c688`	uIP		Integer overflows in IPv6 extension header options.	AFL
`uIP-ext-hdr`	867	`150a3fe–b5d997f`	uIP/RPL*		Unsafe IPv6 extension header processing.	AFL
`uIP-len`	871	`b5d997f–8340735`	uIP	CVE-2020-13985	Unverified IPv6 header length before packet processing.	AFL
`6LoWPAN-frag`	972	`6553688–5884a12`	6LoWPAN		Buffer overflow in 6LoWPAN fragment reassembly.	AFL + external
`SRH-param`	1183	`beff30b–ebd4cae`	RPL*	CVE-2021-21282	Unverified Source Routing Header (SRH) parameter.	Angora + QSym
`ND6-overflow`	1410	`f417a5f–5bfb30d`	IPv6 ND	CVE-2021-21279	Infinite loop in ND6 due to integer wrap around.	QSym
`6LoWPAN-ext-hdr`	1409	`5bfb30d–48a3799`	6LoWPAN	CVE-2021-21280	Out-of-bounds write when processing external header.	Angora + QSym
`SRH-addr-ptr`	1431	`3a3dbfe–3f9a601`	RPL*	CVE-2021-21257	Unverified address pointer in the Source Routing Header.	AFL
`6LoWPAN-decompr`	1482	`425587d–aa6e26f`	6LoWPAN	CVE-2021-21410	Out-of-bounds read when decompressing packets.	MOpt + SymCC
`6LoWPAN-hdr-iphc`	1506	`0dada69–6c8373d`	6LoWPAN		Out-of-bounds read from `hc06_ptr` in a loop condition.	many tools but with ASan
`SNMP-oob-varbinds`	1541	`285cee0–457fa6c`	SNMP		Out-of-bounds read from `varbinds` in a loop condition.	AFL
`SNMP-validate-input`	1517	`457fa6c–9daacb6`	SNMP		Bad length check for SNMP input packets.	AFL
`uIP-RPL-classic-prefix`	1589	`cd208ed–7c2d686`	RPL	CVE-2022-35927	Unverified DIO prefix info lengths.	external
`uIP-RPL-classic-div`	1598	`f608483–e427f48`	RPL		Division by zero from DIO with O lifetimes.	AFL
`6LoWPAN-UDP-hdr`	1646	`b65cfa3–92783e8`	6LoWPAN	CVE-2022-36052	Out-of-bounds read when decompressing UDP header.	MOpt + EffectiveSan
`6LoWPAN-payload`	1647	`92783e8–2dfbaee`	6LoWPAN	CVE-2022-36054	Out-of-bounds write when decompressing payload.	MOpt + EffectiveSan
`uIP-buf-next-hdr`	1648	`2dfbaee–80a5479`	uIP	CVE-2022-36053	Out-of-bounds read in uipbuf.	MOpt + EffectiveSan
`uIP-RPL-classic-sllao`	1654	`8512556–e58b583`	IPv6 ND	CVE-2022-35926	Out-of-bounds read in ND6 option headers.	EffectiveSan into SymCC

first. Once we discovered its root cause, and the Contiki-NG developers applied a fix similar to that of PR#813, all "unique crashes" simply vanished! This taught us something which by now is well-known about fuzzers and their evaluations [11–13]. Namely, that

> *The number of (so called) unique crashes is not a good measure of a fuzzer's efficacy.*

It also taught us something concerning a fuzzer's use. Namely, that one should *stop a fuzzer*—at least all those that rely on AFL—soon after it has come up with the first few "unique crashes." At that point, one should (try to) understand the crashes fuzz testing has exposed, ideally find a fix for them (if they indeed correspond to real bugs in the code), and re-run the fuzzer to check whether similar "unique crashes" show up again or not. More importantly for this paper, our experiences result in a recommendation regarding evaluations and comparisons of fuzzers.

> *Benchmark suites for evaluating fuzzers should try to also capture the process of how bugs are detected and fixed in real software, where often eliminating some bug(s) makes exposition of other bugs a more difficult and time consuming process.*

One natural advantage of such a fuzzer benchmark suite is that it comes with a good definition of what constitutes a bug, and a more accurate indication of how many bugs it contains: the number of pull requests (PRs) that fix a set of related crashes or hangs.

We therefore created a ground truth benchmark suite for fuzzers that comprises the *evolution* of some non-trivial software, in our case Contiki-NG. To that effect, our ground truth suite, currently containing $k = 18$ *known* vulnerabilities, comes with scripts that allow to check out a Git history point of Contiki-NG's development where the first n vulnerabilities are fixed (i.e., *not* there) and the remaining $k - n$ are still present in its code base. Other scripts check whether a particular vulnerability of interest is exposed by a fuzzer during a trial that resulted in crashes or hangs. In short, our ground truth suite *allows for fuzzing experiments focused on some particular vulnerability*. This is exactly what we evaluate in this section: whether fuzzers expose the vulnerability that we are after, and if so how consistently and fast they do this.

Before presenting the results of our evaluation, we mention one additional feature of our ground truth suite, which is related to Contiki-NG's layered architecture. Using only μIP entry point with the RPL-Lite routing protocol did not expose many vulnerabilities. In fact, it resulted in only three PRs; these are shown in the first three rows of Table 2, which fixed vulnerabilities in the code of μIP and RPL protocols. However, by starting the fuzzing at a different layer, namely 6LoWPAN, and changing Contiki-NG configuration, we managed to expose and fix fifteen more vulnerabilities. Finally, note that `6LoWPAN-hdr-iphc` and the last four vulnerabilities in Table 2 were discovered by augmenting fuzzing campaigns with sanitizers.

Vulnerabilities Using µIP as Entry Point. Table 3 shows results of how effective these fuzzers are in exposing bugs in the code of μIP.

For the first two vulnerabilities, we notice the following: (i) The first of them (`uIP-overflow`) is exposed by seven of the eight fuzzers and the second (`uIP-ext-hdr`) is exposed by all fuzzers consistently. (ii) Two of the fuzzers (MOpt and SymCC) are clear winners here in terms of the average time it takes to expose these bugs. (iii) SymCC outperforms QSym, which it uses as a symbolic backend, by a factor similar to the one reported in the SymCC paper [21]. (iv) MOpt's strategy for selecting mutator is particularly effective here; it clearly outperforms the two AFL variants, which are also mutation-based fuzzers, and comes very close to the speed that SymCC achieves.

The third vulnerability (`uIP-len`) is clearly more challenging. First, no fuzzer exposes it in all ten 24-hour trials, and none exposes it fast—or considerably faster than others—either. Second, AFL-cf

Table 3: Number of times and mean time-to-exposure (HH:MM:SS) for the seven vulnerabilities in the code base of μIP.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	**10** 00:17:20	**10** 00:35:40	**10** 00:03:00	0	**10** 00:53:29	**10** 00:23:59	**10** 00:49:58	**10** 00:01:39
uIP-ext-hdr	**10** 03:32:17	**10** 03:23:20	**10** 00:12:11	**10** 00:50:12	**10** 02:44:41	**10** 00:57:23	9 05:05:31	**10** 00:11:35
uIP-len	5 06:59:39	0	4 09:03:11	0	5 08:48:08	5 04:45:32	3 01:24:00	1 01:35:04
uIP-buf-next-hdr	0	0	0	0	0	0	0	0
uIP-RPL-classic-prefix	6 06:21:52	2 18:52:46	7 03:57:22	0	6 09:55:47	**10** 05:14:50	2 07:11:56	0
uIP-RPL-classic-div	7 10:46:12	6 11:09:41	8 07:35:17	4 16:52:41	4 10:54:35	5 08:05:55	3 01:25:26	6 06:00:12
uIP-RPL-classic-sllao	0	0	0	0	0	0	0	0

Table 4: Number of times and mean time-to-exposure for the nine vulnerabilities starting the fuzzing from 6LoWPAN.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
6LoWPAN-frag	**10** 00:17:19	3 12:26:18	**10** 00:12:34	0	**10** 00:18:50	**10** 00:08:32	**10** 00:23:19	**10** 01:40:39
SRH-param	**10** 00:21:23	0	**10** 00:19:44	0	**10** 00:17:09	**10** 00:16:49	**10** 00:18:06	**10** 00:07:34
ND6-overflow	0	0	0	0	0	4 11:15:41	0	0
6LoWPAN-ext-hdr	6 07:13:16	1 13:11:24	**10** 07:52:40	0	9 12:58:16	7 02:36:38	7 08:54:21	3 14:56:08
SRH-addr-ptr	8 02:14:37	0	8 00:31:38	0	7 03:08:56	**10** 00:44:15	8 00:29:51	0
6LoWPAN-decompr	0	0	0	0	0	0	0	0
6LoWPAN-hdr-iphc	0	0	0	0	0	0	0	0
6LoWPAN-UDP-hdr	0	0	0	0	0	0	0	0
6LoWPAN-payload	0	0	0	0	0	0	0	0

never manages to expose it. Quite likely, this is because it uses Clang to instrument the target. A take-away lesson is that *the instrumentation component of a fuzzing tool can cause it to miss vulnerabilities* in addition to influencing the speed of its execution. In retrospect, this should not come as a surprise, because vulnerabilities often exist in code which confuses a compiler or is undefined behavior in the language. In such cases, the compiler is allowed to do whatever it pleases with the code; e.g., even remove it. Another way of stating this lesson is that it is often very difficult to do fair comparisons between different fuzzing tools because they are complex pieces of engineering and assembly of components. For example, SymCC is not just SymCC-QSym (i.e., a fuzzer that uses QSym as a symbolic backend), but is more something like SymCC-QSym-AFL-Clang, and possibly many other parts as well, all of them of particular versions.

Vulnerabilities Using 6LoWPAN as Entry Point. The second set of our evaluation results is shown in Table 4. The first two vulnerabilities (`6LoWPAN-frag` and `SRH-param`) are quite easy to expose for six of the fuzzers (AFL-gcc, MOpt, Angora, QSym, Intriguer, and SymCC), and quite difficult or impossible for AFL-cl and Honggfuzz. However, beyond this point, vulnerabilities become (much) more difficult for most of the fuzzers to expose in 24-hour trials. For example, QSym is the only tool that exposes `ND6-overflow`. (It is also the fuzzer that discovered it.) The next vulnerability, `6LoWPAN-ext-hdr`, is exposed by all but one of the fuzzers. However, note that only MOpt manages to expose it consistently. Similarly, `SRH-addr-ptr` is exposed by five of the eight fuzzers —and by three of them quite fast— but only QSym manages to detect it consistently.

The last vulnerability that we discovered without sanitizers, `6LoWPAN-decompr`, is not exposed by any of the fuzzers in any of the ten 24-hour experiments that we ran to record timing measurements. For this vulnerability, we executed some more trials for all fuzzers —some of them longer than two days— but they did not expose it either. However, as also shown in Table 2, this vulnerability was originally discovered by both MOpt and SymCC, *without* the use of sanitizers, so at least these two fuzzers *could* have exposed it. (Note that for each vulnerability of interest, we use the Git commit SHA *before* the PR that fixed it was merged in the code base.)

We mention that our suites come with Proof of Vulnerability (PoV) support, in the form of inputs that trigger crashes or hangs and scripts to trigger them. For example, the `6LoWPAN-decompr` vulnerability is easily triggered when Contiki-NG's code is instrumented with AddressSanitizer [24].

Answers to RQ.1–RQ.3

Regarding RQ.1, our results do show some advantage of using hybrid fuzzers in complex code bases such as Contiki-NG's network stack compared to mutation-based fuzzers, but do not show any clear superiority of hybrid fuzzing techniques. We also discover that no fuzzer is uniformly better than all the rest in terms of exposing more vulnerabilities. Still, there are three fuzzers that stand out among the eight we consider: MOpt, SymCC, and QSym. Besides exposing bugs faster than the rest (RQ.2), they are the fuzzers that expose bugs more consistently (RQ.3) among them; cf the entries of Tables 3 and 4.

We mention that these results agree with the conclusions of the UniFuzz paper [13], which also considers MOpt and QSym (but not SymCC) in its evaluation and places them on top. However, it differs from a recently published paper on fuzzing effectiveness using FuzzBench [2], which also considers MOpt, but places AFL++ and Honggfuzz on top. Our work does not consider AFL++, so we cannot say anything about this fuzzer, but on Contiki-NG's code base Honggfuzz does not perform well. We were curious about this result and also tried Honggfuzz in configurations with many threads in its thread pool, but the result did not change.

Let us highlight two reasons why, in our opinion, hybrid fuzzers are not more effective on Contiki-NG's codebase.

First, hybrid fuzzers tend to restrain themselves in doing two things: (1) getting seeds from coverage-guided grey-box fuzzers, and (2) applying heavier analyses to produce better mutants. Doing so, hybrid fuzzers heavily rely on mutation-based fuzzers, and get stuck whenever these cannot generate any new mutants. In short:

> *The consistency and effectiveness of a hybrid fuzzer is dependent on the consistency and effectiveness of its mutation-based component.*

Table 5: Number of times and mean time-to-exposure for the μIP vulnerabilities using AddressSanitizer instrumentation.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	8 00:17:24	**10** 00:34:34	**10** 00:19:53	0 🕒	**10** 00:48:04	**10** 00:15:08	**10** 00:37:30	**10** 00:31:03
uIP-ext-hdr	**10** 05:15:10	**10** 02:30:14	**10** 01:20:44	**10** 01:11:22	**10** 02:17:21	**10** 01:53:00	**10** 03:33:16	**10** 02:38:00
uIP-len	0 🕒	0 🕒	0 🕒	0 🕒	0 🕒	0 🕒	0 🕒	2 11:57:49
uIP-RPL-classic-prefix	2 13:25:17	0 🕒	2 21:58:18	0 🕒	1 03:59:56	1 08:19:18	0 🕒	1 17:06:14
uIP-RPL-classic-div	0 🕒	0 🕒	0 🕒	2 09:50:03	1 02:41:05	0 🕒	0 🕒	0 🕒

Table 6: Impact of AddressSanitizer for the vulnerabilities in the code base of μIP. The table shows performance differences from Table 3: a positive impact is denoted with an upward arrow (▲) and negative impact with a downward arrow (▼). An integer denotes the change in the number of trials exposing the vulnerability; for similar number of trials the time difference is shown. A number of trials and a time denote vulnerabilities that a fuzzing tool exposed only on the sanitized code.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	▼ 2	▲ 00:01:06	▼ 00:16:53	–	▲ 00:05:25	▲ 00:08:51	▲ 00:12:28	▼ 00:29:24
uIP-ext-hdr	▼ 01:42:53	▲ 00:53:06	▼ 01:08:33	▼ 00:21:10	▲ 00:27:20	▼ 00:55:37	▲ 1	▼ 02:26:25
uIP-len	▼ 5	–	▼ 4	–	▼ 5	▼ 5	▼ 3	▲ 1
uIP-RPL-classic-prefix	▼ 4	▼ 2	▼ 5	–	▼ 5	▼ 9	▼ 2	▲ 1
uIP-RPL-classic-div	▼ 7	▼ 6	▼ 8	▼ 2	▼ 3	▼ 5	▼ 3	▼ 6

An idea that follows naturally from this observation is that hybrid fuzzers can consider integration with multiple mutation-based fuzzers to mitigate this dependency.

Second, the symbolic/concolic execution component of a hybrid fuzzer focuses on generating mutants that penetrate new code blocks or explore new paths (i.e., on increasing coverage) rather than mutants that have increased changes to detect more bugs.

Regarding this second point, sanitizers can in principle aid a fuzzer's dynamic analysis to expose difficult to find bugs. To investigate this point, we augment our fuzzing experiments with sanitizer instrumentation in the next section.

4 IMPACT OF SANITIZERS

Sanitizers are dynamic bug-finding tools that analyze a single program execution using different types of instrumentation. In theory, sanitizers can significantly increase the defect detection capability of fuzzers [23, 25]. On the other hand, sanitizers impose a non-negligible execution overhead, due to code instrumentation. Thus, sanitizers represent a trade-off as far as fuzz testing is concerned, since fuzzing tools rely on their high execution speed (esp. compared to other testing techniques such as symbolic execution) to be able to execute the SUT with a large number of inputs.

In this section, we investigate RQ.4. To answer that question, we evaluate the tools on the same set of vulnerabilities using two sanitizers: AddressSanitizer [24] (ASan) and Effective Type Sanitizer [7] (EffectiveSan). More precisely, we add sanitizer instrumentation during the compilation of the coverage-guided, grey-box targets (i.e., the binaries used by AFL and MOpt). Then, as in §3, we run ten fuzzing campaigns for every tool feeding them those augmented targets. Following the usual methodology when fuzzing, we configure the sanitizers to crash whenever they detect a suspicious input, which consequently alerts the fuzzer of the error.

As in the experiments of §3, we report the number of trials exposing a witness with the mean time-to-exposure (TTE). To ease the comparison between fuzzing without and with the aid of a sanitizer, we include a second table focusing on the tools' difference from the results in Tables 3 and 4. This table depicts an integer if a different number of trials are exposing the vulnerability or the added mean TTE in case of similar effectiveness. Due to space limitations, we do not present results for cases where vulnerabilities are not discovered with or without sanitizers (as e.g., is the case for `uIP-buf-next-hdr` and `uIP-RPL-classic-sllao`), but these results can be found in the paper's artifact. For the vulnerabilities exposed by the sanitizers only, we report the number of trials together with the mean TTE. Finally, we denote a positive sanitizer impact, i.e., more trials exposing the vulnerability or a shorter mean TTE, using upper arrows (▲). Down arrows (▼) denote a negative impact.

4.1 Fuzzing with AddressSanitizer

AddressSanitizer [24] (ASan) is a popular sanitizer adding extra memory areas, called red zones, around memory allocations. These red zones dynamically detect any invalid dereferences that are pointing within such addresses. ASan's technique is not complete and might miss some dereferences that do not point to a red zone, but it improves the exposure of out-of-bound memory access errors.

We investigate the impact of ASan on the fuzzing tools in exposing vulnerabilities in Contiki-NG's code base. To combine the sanitizer with AFL's instrumentation, we use the environment variable provided by the companion scripts, passing sanitizer options to the compiler. At runtime, ASan requires disabling the memory limit and increasing the timeout as its initialization pre-allocates a consequent chunk of memory, causing the AFL to stop. When fuzzing the Contiki-NG's code base, AFL reports only 100 exec/s compared to 1 200 exec/s for the fuzzing campaigns in §3, i.e., a runtime overhead of more than 10×. Tables 5 and 7 show the tools performance with ASan instrumentation. The differences with Tables 3 and 4 are shown in Tables 6 and 8, respectively.

Vulnerabilities Using μIP as Entry Point. Tables 5 and 6 show a fair impact for the two first vulnerabilities. With ASan instrumentation, AFL-gcc exposes `uIP-overflow` in two trials fewer than when running without a sanitizer showing a noticeable drop. SymCC, which is another fuzzer negatively affected by the sanitizer overhead, now requires on average thirty minutes instead of less than two to expose `uIP-overflow`. On the other hand, Intriger, Angora, and AFL-cf expose the `uIP-ext-hdr` vulnerability in more trials or

Table 7: Number of times and mean time-to-exposure for the 6LoWPAN vulnerabilities and AddressSanitizer instrumentation.

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
6LoWPAN-frag	9 01:39:07	8 01:06:08	8 02:19:24	0	**10** 00:28:16	**10** 00:31:59	**10** 02:03:18	**10** 00:40:38
SRH-param	0	0	0	0	0	0	0	0
6LoWPAN-ext-hdr	**10** 01:01:17	**10** 01:11:11	**10** 00:18:16	**10** 06:59:36	**10** 00:36:27	**10** 00:16:53	**10** 01:16:03	**10** 00:39:00
SRH-addr-ptr	0	0	0	0	0	0	0	4 03:37:12
6LoWPAN-decompr	**10** 00:03:25	**10** 00:03:15	**10** 00:01:45	**10** 00:00:19	**10** 00:02:07	**10** 00:02:26	**10** 00:03:00	**10** 00:01:19
6LoWPAN-hdr-iphc	9 08:38:23	**10** 06:21:53	**10** 03:19:03	**10** 02:00:48	8 07:57:00	8 03:32:29	9 09:28:19	9 10:04:55

Table 8: Impact of AddressSanitizer for the vulnerabilities starting the fuzzing from 6LoWPAN (differences from Table 4).

Id	AFL-gcc	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
6LoWPAN-frag	▼ 1	▲ 5	▼ 2	—	▼ 00:09:26	▼ 00:23:27	▼ 01:39:59	▲ 01:00:01
SRH-param	▼ 10	—	▼ 10	—	▼ 10	▼ 10	▼ 10	▼ 10
6LoWPAN-ext-hdr	▲ 4	▲ 9	▲ 07:34:24	▲ 10	▲ 1	▲ 3	▲ 3	▲ 7
SRH-addr-ptr	▼ 8	—	▼ 8	—	▼ 7	▼ 10	▼ 8	▲ 4
6LoWPAN-decompr	▲ 10	▲ 10	▲ 10	▲ 10	▲ 10	▲ 10	▲ 10	▲ 10
6LoWPAN-hdr-iphc	▲ 9	▲ 10	▲ 10	▲ 10	▲ 8	▲ 8	▲ 9	▲ 9

faster on average, whereas the other tools have only slight slowdowns. The impact is very negative for `uIP-len`. All the tools but SymCC failed to expose the vulnerability. Notice that SymCC uses the most recent compiler within the tools' configurations, and we experienced a better detection of the vulnerability with ASan and recent Clang. None of the tools exposed `uIP-buf-next-hdr`, a vulnerability which has been discovered using Effective Type Sanitizer. Moreover, only a few fuzzers expose `uIP-RPL-classic-prefix`, and this happens only once or twice.

Vulnerabilities Using 6LoWPAN as Entry Point. Next, we look at Tables 7 and 8. AFL-cf and SymCC expose `6LoWPAN-frag` clearly better. Interestingly, the impact of ASan is positive only for the tools based on the Clang compiler. For `SRH-param`, none of the tools exposes the vulnerability with ASan instrumentation, and later, we see similar results for `SRH-addr-ptr`. Note that ASan's red zones algorithm is known to be incomplete, and such behaviors may be due to this incompleteness. On the other hand, ASan's impact is positive on all the tools in exposing the three following vulnerabilities. All tools consistently expose `6LoWPAN-ext-hdr` and `6LoWPAN-decompr`, and pretty fast for the latter. Notice that the former vulnerability is quite challenging, and none of the tools found a witness without ASan for the latter. The `6LoWPAN-hdr-iphc` vulnerability has been discovered by adding ASan instrumentation to our fuzzing campaigns, but only AFL-cf, and MOpt are exposing it consistently.

4.2 Fuzzing with Effective Type Sanitizer

Effective Type Sanitizer [7] (EffectiveSan) implements a dynamic type checking technique that tracks type values during the execution, preserving static high-level C/C++ type information and employing extended pointers. As a result, this sanitizer checks: 1) whether any dynamic type matches its corresponding static type, and 2) in case of pointers, whether the offset is within bounds. This technique is particularly efficient to detect out-of-bound memory accesses based on object boundaries, which can happen when accessing buffers without proper offset validation.

EffectiveSan and AFL Configuration. EffectiveSan is available as an extension of Clang-4.0.1. Consequently, we cannot use AFL-gcc companion scripts anymore and need to change several configurations of Table 1: (1) we use AFL-clang instead of AFL-gcc instrumentation, and (2) we set EffectiveSan's Clang to apply both AFL and EffectiveSan instrumentation during the compilation of the augmented targets. (For SymCC in Ubuntu-20, we compile Clang with gcc-4.8.2.) Unfortunately, we had an issue combining AFL-clang-fast and EffectiveSan, due to an AFL's global variable. To fix the issue, we had to disable the EffectiveSan's track of global variables, leaving EffectiveSan with a lighter analysis. Notice that EffectiveSan does not require heavy memory allocations, and AFL can fuzz the target without increasing its usual limits. As a consequence, AFL reports between 100 and 500 exec/s, giving a better throughput than when using ASan. Tables 9 and 11 (on top of the next page) depict the tools' performance when fuzzing with EffectiveSan instrumentation. Differences with Tables 3 and 4 are shown in Tables 10 and 12.

Vulnerabilities Using µIP as Entry Point. Compared to Table 3, Tables 9 and 10 show impressive results. Six of the eight fuzzers now consistently expose the three first vulnerabilities with EffectiveSan, often with better average time-to-exposures than the ones in Table 3. Furthermore, for all fuzzers except AFL-clang-fast, there is a remarkable improvement in exposing the `uIP-len` vulnerability. Like in the ASan campaigns, there is a drop in the number of trials that expose the bugs for both vulnerabilities on the RPL-Classic routing protocol. However, we can also notice that SymCC shows a better exposure of `uIP-RPL-classic-prefix`. Notice the AFL-cf exception; disabling the sanitizer track of global variables is apparently affecting the results in this case. Finally, the tools are exposing `uIP-buf-next-hdr` with EffectiveSan. However, the vulnerability is still challenging to expose consistently, though we notice increased effectiveness and better performance for SymCC.

Vulnerabilities Using 6LoWPAN as Entry Point. There is also good news with the vulnerabilities in Tables 11 and 12. Almost all the tools expose all six of the nine vulnerabilities. (Compared to Table 4 there are three vulnerabilities that still remain unexposed.) Angora managed to expose `SRH-param` in only seven of its ten trials. Only MOpt and QSym managed to expose the most challenging vulnerability (`6LoWPAN-frag`), and did that only once, when running with sanitizer instrumentation.

4.3 Feeding Corpora of Fuzzers to Sanitizers

In the previous experiments, we injected sanitizer instrumentation during the compilation of the mutation-based fuzzers' targets and

Table 9: Number of times and mean time-to-exposure for the μIP vulnerabilities and EffectiveSan instrumentation.

Id	AFL-clang	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	**10** 00:10:49	**10** 00:09:19	**10** 00:16:19	0 ◷	**10** 00:06:07	**10** 00:14:56	**10** 00:20:15	**10** 00:05:52
uIP-ext-hdr	**10** 01:03:07	**10** 03:35:05	**10** 00:24:04	0 ◷	**10** 00:42:26	**10** 01:15:08	**10** 00:35:02	**10** 00:24:05
uIP-len	**10** 00:44:24	0 ◷	**10** 00:25:34	**10** 04:06:35	**10** 02:29:14	**10** 02:02:46	**10** 02:01:25	**10** 00:17:42
uIP-buf-next-hdr	2 12:47:46	0 ◷	3 08:36:57	0 ◷	1 01:52:32	2 00:29:24	2 07:13:00	7 06:41:59
uIP-RPL-classic-prefix	0 ◷	0 ◷	3 13:28:30	0 ◷	0 ◷	2 04:51:57	2 13:23:10	5 03:22:02
uIP-RPL-classic-div	3 22:04:40	0 ◷	3 19:27:27	0 ◷	2 04:29:34	1 02:31:07	2 18:44:25	6 08:53:54

Table 10: Impact of EffectiveSan for the vulnerabilities in the code base of μIP (differences from Table 3).

Id	AFL-gcc/-clang	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
uIP-overflow	▲ 00:06:31	▲ 00:26:21	▼ 00:13:19	—	▲ 00:47:22	▲ 00:09:03	▲ 00:29:43	▼ 00:04:13
uIP-ext-hdr	▲ 02:29:10	▼ 00:11:45	▼ 00:11:53	▼ 10	▲ 02:02:15	▼ 00:17:45	▲ 1	▼ 00:12:30
uIP-len	▲ 5	—	▲ 6	▲ 10	▲ 5	▲ 5	▲ 7	▲ 9
uIP-buf-next-hdr	▲ 2	—	▲ 3	—	▲ 1	▲ 2	▲ 2	▲ 7
uIP-RPL-classic-prefix	▼ 6	▼ 2	▼ 4	—	▼ 6	▼ 8	▼ 06:11:14	▲ 5
uIP-RPL-classic-div	▼ 4	▼ 6	▼ 5	▼ 2	▼ 2	▼ 4	▼ 1	▼ 02:53:42

Table 11: Number of times and mean time-to-exposure for the 6LoWPAN vulnerabilities and EffectiveSan instrumentation.

Id	AFL-clang	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
6LoWPAN-frag	0 ◷	1 08:50:49	1 00:11:43	0 ◷	0 ◷	1 00:03:15	0 ◷	0 ◷
SRH-param	**10** 02:13:48	0 ◷	**10** 02:30:14	0 ◷	7 01:21:25	**10** 00:25:01	**10** 01:29:04	**10** 00:09:34
6LoWPAN-ext-hdr	**10** 00:05:03	0 ◷	**10** 00:03:35	0 ◷	**10** 00:11:24	**10** 00:09:04	**10** 00:10:14	**10** 00:06:12
SRH-addr-ptr	**10** 02:40:11	0 ◷	9 01:24:37	0 ◷	9 04:40:44	9 00:51:55	**10** 02:08:22	**10** 00:15:36
6LoWPAN-decompr	**10** 00:00:48	0 ◷	**10** 00:00:33	**10** 00:06:04	**10** 00:02:09	**10** 00:01:07	**10** 00:00:50	**10** 00:00:20
6LoWPAN-hdr-iphc	**10** 01:58:58	0 ◷	**10** 02:26:18	1 08:30:50	**10** 07:18:57	**10** 07:26:44	**10** 04:44:29	9 02:14:19
6LoWPAN-payload	**10** 00:10:49	0 ◷	**10** 00:04:19	0 ◷	**10** 02:00:20	**10** 00:41:21	**10** 01:00:01	**10** 00:15:16

Table 12: Impact of EffectiveSan for the vulnerabilities starting the fuzzing from 6LoWPAN (differences from Table 4).

Id	AFL-gcc/-clang	AFL-cf	MOpt	Honggfuzz	Angora	QSym	Intriguer	SymCC
6LoWPAN-frag	▼ 10	▼ 2	▼ 9	—	▼ 10	▼ 9	▼ 10	▼ 10
SRH-param	▼ 01:52:25	—	▼ 02:10:30	—	▼ 3	▼ 00:08:12	▼ 01:10:58	▼ 00:02:00
6LoWPAN-ext-hdr	▲ 4	▼ 1	▲ 07:49:05	—	▲ 1	▲ 3	▲ 3	▲ 7
SRH-addr-ptr	▲ 2	—	▲ 1	—	▲ 2	▼ 1	▲ 2	▲ 10
6LoWPAN-decompr	▲ 10	—	▲ 10	▲ 10	▲ 10	▲ 10	▲ 10	▲ 10
6LoWPAN-hdr-iphc	▲ 10	—	▲ 10	▲ 1	▲ 10	▲ 10	▲ 10	▲ 9
6LoWPAN-payload	▲ 10	—	▲ 10	—	▲ 10	▲ 10	▲ 10	▲ 10

launched fuzzing campaigns with those targets. In this final experiment, we employ corpora (the good and bad inputs) generated from the ten trials of §3 and feed these inputs to the mutation-based fuzzers' target augmented with sanitizers. In other words, we apply sanitizer analyses afterward without launching a new campaign but we use the mutants generated by a previous one (and without sanitizer instrumentation). Consequently, for a time budget of 24 hours, we compare the method of using sanitizers within fuzzing campaigns (henceforth denoted as Method A), as is done in §4.1 and §4.2, against the method of using sanitizers on the output queue generated from such fuzzing campaigns (denoted as Method B). For the evaluation, we use the same binaries: the SUT compiled with both fuzzer and sanitizer instrumentation. Tables 13 and 14 report selected results, showing the number of trials detecting a witness for the hard to detect vulnerabilities of Tables 3 and 4. (The complete set of results can be found in the paper's artifact.)

Feeding Corpora to ASan. Table 13 shows the number of trials exposing vulnerabilities using both methods for ASan. As a general comment, effectiveness does change for five out of the ten hard-to-expose vulnerabilities with ASan. Most of the tools do not expose any of the two first vulnerabilities. That is not a surprise for `uIP-buf-next-hdr`, which has been discovered by EffectiveSan. However, regarding `uIP-len`, almost all the witnesses from §3 are

Table 13: Number of times the targets with AFL and ASan instrumentation expose the challenging vulnerabilities from §4.1. On the left, we show the trials of §4.1, i.e., during ASan campaigns. On the right, we depict the trials exposing the vulnerability by feeding corpora of §3 to the targets.

Id	*AFL-gcc*	*AFL-cf*	*MOpt*	*Angora*	*QSym*	*Intriguer*	*SymCC*
uIP-len	0 0	0 0	0 0	0 0	0 0	0 0	2 1
uIP-buf-next-hdr	0 0	0 0	0 0	0 0	0 0	0 0	0 0
uIP-RPL-classic-prefix	2 6	0 2	2 7	1 6	1 10	0 2	1 2
uIP-RPL-classic-div	0 7	0 6	0 8	1 4	0 5	0 3	0 6
uIP-RPL-classic-sllao	0 0	0 0	0 0	0 0	0 0	0 0	0 0
SRH-param	0 0	0 0	0 0	0 0	0 0	0 0	0 10
ND6-overflow	0 0	0 0	0 0	0 0	0 4	0 0	0 0
SRH-addr-ptr	0 0	0 0	0 0	0 0	0 0	0 0	4 0
6LoWPAN-UDP-hdr	0 0	0 0	0 0	0 0	0 0	0 0	0 0
6LoWPAN-payload	0 0	0 0	0 0	0 0	0 0	0 0	0 0

not detected anymore by adding ASan to the targets. That is surprising and tells us that fuzzing with ASan can make the exposure of vulnerabilities (when staying within a time budget) actually more difficult due to the runtime overhead that its instrumentation adds. Exposing the two first vulnerabilities using μIP entry point and the RPL-Classic routing protocol is not a problem anymore following Method B. Notice, though, that the number of trials did not

Table 14: Similar to Table 13 but with EffectiveSan.

Id	*AFL-cl*	*AFL-cf*	*MOpt*	*Angora*	*QSym*	*Intriguer*	*SymCC*
uIP-buf-next-hdr	2 0	0 0	3 0	1 0	2 2	2 0	7 0
uIP-RPL-classic-prefix	0 6	0 2	3 7	0 6	2 10	2 2	5 2
uIP-RPL-classic-div	3 7	0 6	3 8	2 4	1 5	2 3	6 6
uIP-RPL-classic-sllao	0 0	0 0	0 0	0 0	0 0	0 0	0 0
6LoWPAN-frag	0 1	1 3	1 0	0 0	1 2	0 1	0 0
ND6-overflow	0 0	0 0	0 0	0 0	1 4	0 0	0 0
6LoWPAN-UDP-hdr	0 0	0 0	0 0	0 0	0 0	0 0	0 0

change at all: adding ASan did not expose other bad inputs missed during §3 experiments.

Let us now focus on the vulnerabilities starting with 6LoWPAN entry point. Though there are witnesses in Table 4, `SRH-param` and `SRH-addr-ptr` are not exposed following Method B either. Except for SymCC, which has a more recent configuration, those behaviors are similar to the ones we have seen for `uIP-len`: the ASan instrumentation prevents the fuzzer from exposing those vulnerabilities.

For the vulnerability `SRH-addr-ptr`, which is exposed better by SymCC following Method A, we must launch the fuzzing campaigns with sanitizer instrumentation to expose it. To understand the reason, let us assume that a fuzzer has just generated an input witnessing `SRH-addr-ptr`. If this input neither crashes—because the sanitizer instrumentation is not enabled—nor provides new coverage, the fuzzer will discard it. Consequently, the witness is lost and even after feeding the corpus to ASan, the tools will not expose the vulnerability. We call this kind of vulnerability *path-sensitive*. Path-sensitive vulnerabilities are interesting because they require a good analysis during the fuzzing campaigns to be exposed.

Feeding Corpora to EffectiveSan. Table 14 depicts the number of trials exposing the vulnerabilities of Tables 3 and 4 that were not reported in §4.2. Surprisingly, following Method B, none of the tools expose `uIP-buf-next-hdr`, even though it was originally discovered by EffectiveSan. We have good reasons to believe that the vulnerability is path-sensitive (like `SRH-addr-ptr` with ASan, its witnesses are discarded by the coverage-guided grey-box component). With the exception of `uIP-RPL-classic-sllao`, the tools expose better the vulnerabilities on the code of RPL-Classic routing protocol. Also, first running standard fuzzing campaigns and then feeding the corpus to EffectiveSan makes the tools expose `uIP-RPL-classic-prefix` and `uIP-RPL-classic-div` in almost all the cases. The `uIP-RPL-classic-sllao` vulnerability is still undetected due to a conflict with AFL instrumentation. Indeed, only after running the same files with EffectiveSan instrumentation, could we expose `uIP-RPL-classic-sllao` witnesses.

Let us now look at 6LoWPAN vulnerabilities, in particular the `6LoWPAN-frag` one, which is hardly exposed even following Method B. In fact, due to EffectiveSan, most of the previous witnesses are still crashing after the fixes. Hence, it is much more difficult for the fuzzers to find an input fixed by the `6LoWPAN-frag` pull request. This case highlights the potential conflicts other bugs have on exposing a specific vulnerability even with a heavier analysis. For the last two vulnerabilities, there are no surprises: QSym still exposes `ND6-overflow` in four of the ten trials, and none of tools exposes `6LoWPAN-UDP-hdr`.

Answer to RQ.4

From our experiments, we find that, overall, ASan and EffectiveSan pay off for their overhead. Sanitizers are essential for exposing five vulnerabilities that cannot be detected without sanitizer use, and the eight tools expose Contiki-NG vulnerabilities more consistently. Furthermore, we see that using sanitizers afterward can expose deep vulnerabilities, but not path-sensitive ones. The tools expose the latter only during fuzzing campaigns with sanitizer instrumentation enabled. This point should motivate the fuzzing community to improve fuzzers' analyses during target execution.

As a last comment, note that what we do *not* know whether Contiki-NG's code base at the point where the last vulnerability we have detected (`uIP-RPL-classic-sllao`) is fixed does not contain any *other* vulnerabilities that the fuzzers and sanitizers we have used so far cannot expose. On the other hand, we see this as a positive feature and an opportunity to expand our ground truth suite with more vulnerabilities and more fuzzing tools in the future, especially more powerful ones than the ones we have selected for this paper.

5 RELATED WORK

In recent years, several efforts have been made to evaluate and compare the performance of different fuzz testing tools on real-world software. In this section, we cover some related work on fuzzer benchmark suites and comparative evaluations of fuzzers.

Commonly Used Benchmark Suites. LAVA-M [6] and the DARPA Cyber Grand Challenge (CGC) [3] benchmark suites are the first to propose a common set of buggy software for evaluating fuzzers. LAVA-M automatically injects numerous out-of-bounds memory accesses into programs in coreutils-8.14. However, all bugs are triggered by "magic value" comparisons, which does not accurately represent the complexity and diversity of software bugs encountered in the real-world. The DARPA CGC bug suite proposes a wider range of bug types, but its synthetic programs are relatively simple and small. In contrast, our work offers a ground truth benchmark suite for fuzzers using the Contiki-NG network stack code base. We built that suite from real vulnerabilities, which correspond to several different types of software errors. All the bugs come with crashing inputs (aka proof of vulnerability), fixes that correct them, and capture different time points in the evolution of Contiki-NG's code base. Also, our suite comes with bugs that have a natural progression in the level of difficulty to detect them, something which is missing in all existing benchmark suites for evaluating fuzzers.

Google FuzzBench [10], previously Google Fuzzer Test Suite [9], is an online service to evaluate fuzzers. FuzzBench proposes an easy integration of new fuzzers together with a periodic evaluation of them on a set of benchmarks. The service makes it easier for fuzzer developers to evaluate some performance aspects of their tools, but reports only code coverage statistics which, although indicative, *overapproximate* bug coverage and are insufficient to compare the effectiveness of different fuzzing techniques and tools (i.e., a fuzzer should not only reach the statement where a bug exists, but it also needs to *expose* that bug).

Two recent efforts, running concurrently with our work, have proposed benchmarking platforms with different characteristics, both between them and from the suite we have put forward.

UniFuzz. The first effort, UniFuzz, offers a *"holistic and pragmatic metrics-driven platform for evaluating fuzzers,"* currently incorporating numerous existing fuzzers and 20 real-world programs. Clearly it is the result of significant effort, and provides a platform which is much broader than ours. Similarly to what we do, UniFuzz comes with support for Docker containers to easily reproduce experiments and to add new fuzzers if desired. However, unlike our suites, UniFuzz triages a fuzzer's observed crashes by mapping crash stack traces to known CVEs, which is sometimes problematic in our opinion, and does not explicitly specify the number of bugs a fuzzer is expected to find (i.e., the UniFuzz suite lacks ground truth knowledge). It is also unclear whether UniFuzz allows for trials that focus only on specific (subset of) bugs. Finally, UniFuzz does not allow to test the *evolution* of the target programs, an aspect we consider important in fuzzer comparisons (e.g., for determining fuzzers that stop being effective beyond some particular point).

The UniFuzz paper [13] evaluates eight fuzzers (AFL, AFLFast, Angora, Honggfuzz, MOpt, QSym, T-Fuzz, and VUzzer64), i.e., has five fuzzers in common with our evaluation. Similar to our results, its evaluation also finds that none of these fuzzers outperforms all the others across all target programs, but that MOpt and QSym are in the top category, a statement that our results also corroborate.

Magma. The second effort, Magma [11], provides a *"ground truth fuzzing benchmark that enables fuzzing evolution and comparison."* Magma incorporates a large diversity of programs and bug types. One of its key design decisions is to *forward-port* pull requests that fix bugs into the latest version of a program's code together with the condition to trigger them. This is a very interesting idea, which is complementary and 'in the other direction' to what we do in our ground truth suite. The bug conditions allow Magma to monitor whether a fuzzer has triggered a bug, but was unable to detect it (i.e., if the bug condition is satisfied but the target did not crash). This metric nicely refines the runtime information provided by the benchmark during fuzzing. However, not all the forward-ported bugs have an updated proof of vulnerability, and some may not be possible to trigger in the latest version of the program's code. Furthermore, the monitoring instrumentation adds restrictions on how fuzzers should execute a target.

The Magma paper [11] evaluates seven fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, Honggfuzz, and SymCC) on the Magma suite using an extensive set of trial runs. The fuzzing tools in common to those we use in this paper is three mutation-based (AFL, MOpt-AFL, and Honggfuzz) and only one hybrid fuzzer (SymCC).

In contrast to Magma, in our suite we do not port the bugs into a different context. Instead, we directly checkout to the corresponding Git history points. Every vulnerability has a pair of associated commits: those before and after the bug fix. Using this pair of commits, we can ensure the bug reproducibility and provide the *exact same environment and conditions* for bug detection.

Evaluation of Fuzz Testing. Klees *et al.* warn that missing the fuzzer's randomness factor leads to mis-interpretation during experiments [12]. Furthermore, their paper shows the bias of using crash-based metrics, and the authors argue that the community should use bug metrics with statistical relevance instead. We have followed this methodology by running our experiments ten times with a timeout of 24 hours. Moreover, when fuzzers find the bug in all trials, we compute the Mann-Whitney significance of the corresponding best time-to-exposure. However, due to lack of space, this data is only available in our artifact.

6 CONCLUDING REMARKS

There exist at least two different ways to read this paper. The first, perhaps less exciting one, is to view it as an experience report of using different state-of-the-art fuzzing tools to detect vulnerabilities in the complex code base of a widely-used OS for IoT devices, and improve its robustness and security. In this respect, our advice to other developers is: "*use as many fuzzers as you can get your hands to and fix, or at least try to understand, the issues that they report.*" We offer some strong evidence that, currently, no single fuzzing tool outperforms all the others or is able to consistently expose the bugs that exist or, worse, that itself has previously discovered. In our opinion, this *calls for research that makes fuzzers more consistent*, besides making them faster and/or more effective.

Another way to read this paper is as offering an independent and extensive evaluation of the effectiveness of state-of-the-art mutation-based and hybrid fuzzers on a real-world code base. It also proposes a new benchmark suite for evaluating fuzzers, which has special properties due to the layered-based characteristics of Contiki-NG's code base. Finally, it offers some new ideas on how fuzzing tools should be evaluated. Evaluating fuzz testing tools accurately and consistently is not an easy task, and will remain challenging as techniques mature and get incorporated into tools that come with more and more knobs, bells and whistles. We hope that our benchmarking platform proves useful to researchers and developers of the area, and that it will get extended as more vulnerabilities are exposed in Contiki-NG's code base.

As a final comment, we note that none of the fuzzers we have used in this evaluation takes into account the stateful nature of the protocols implemented in Contiki-NG's low-power IPv6 stack. When we started this work, *stateful grey-box fuzzers* were nonexistent (or still in their infancy), but since then some such fuzzers have been developed. Extending our experiments with a set of stateful fuzzers and discovering whether the known bugs are exposed faster, more consistently, or measure how many more new bugs such fuzzers are able to expose are intesting directions for continuing this work. Similarly, one can include *ensemble fuzzers* in a future comparison. However, as noted in the paper's title, there are so many fuzzers and so little time...

ACKNOWLEDGMENTS

This research has been supported in part by the Swedish Foundation for Strategic Research through the aSSIsT project and by the Swedish Research Council through grant #621-2017-04812. We thank the anonymous reviewers for their time and their comments.

REFERENCES

[1] AFL-clang-fast 2019. Fast LLVM-based instrumentation for afl-fuzz. https://github.com/google/AFL/tree/master/llvm_mode.

[2] Dario Asprone, Jonathan Metzman, Arya Abhishek, Guizzo Giovani, and Federica Sarro. 2022. Comparing Fuzzers on a Level Playing Field with FuzzBench. In *15th IEEE International Conference on Software Testing, Verification and Validation* (Valencia, Spain) *(ICST 2022)*. IEEE, 302–311. https://doi.org/10.1109/ICST53961.2022.00039

[3] Brian Caswell. 2016. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/.

[4] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy* (San Francisco, CA) *(SP 2018)*. IEEE, 711–725. https://doi.org/10.1109/SP.2018.00046

[5] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, UK) *(CCS '19)*. ACM, New York, NY, USA, 515–530. https://doi.org/10.1145/3319535.3354249

[6] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy* (San Jose, CA, USA) *(SP 2016)*. IEEE Computer Society, 110–121. https://doi.org/10.1109/SP.2016.15

[7] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 181–195. https://doi.org/10.1145/3192366.3192388

[8] Adam Dunkels. 2003. Full TCP/IP for 8-Bit Architectures. In *1st International Conference on Mobile Systems, Applications and Services* (San Francisco, CA, USA) *(MobiSys 2003)*. USENIX, 85–98. https://doi.org/10.1145/1066116.1066118

[9] FTS 2018. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.

[10] FuzzBench 2020. FuzzBench: Fuzzer Benchmarking As a Service. https://github.com/google/fuzzbench.

[11] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428334

[12] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[13] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[14] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, and Raheem Beyah. 2019. MOpt: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, USA). USENIX Association, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[15] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[16] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[17] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. 2022. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX* 18 (2022), 101089. https://doi.org/10.1016/j.softx.2022.101089

[18] Brian S. Pak. 2012. *Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution.* Master's thesis. School of Computer Science, Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/2012/CMU-CS-12-116.pdf CMU-CS-12-116.

[19] Mathias Payer. 2019. The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes. *IEEE Security & Privacy* 17, 1 (Jan.–Feb. 2019), 78–82. https://doi.org/10.1109/MSEC.2018.2889892

[20] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC'19)*. ACM, New York, NY, USA, 163–176. https://doi.org/10.1145/3359789.3359796

[21] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)* (Boston, MA, USA). USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[22] RFC 2007. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. https://tools.ietf.org/html/rfc4944.

[23] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code. Talk at *USENIX Enigma*. https://www.usenix.org/conference/enigma2016/conference-program/presentation/serebryany

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, USA). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[25] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *IEEE Symposium on Security and Privacy (SP 2019)*. IEEE, USA, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[26] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS 2016)*. The Internet Society, 16 pages. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[27] Robert Swiecki. 2010. Honggfuzz. https://honggfuzz.dev/.

[28] The Clang Team. 2014. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[29] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, USA). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[30] Michał Zalewski. 2013. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.