Brief paper summary (2-3 sentences)

In this paper the authors introduce AFL-IoT -- a port of the popular Linux-based coverage-guided fuzzer AFL to popular Linux-based IoT devices. The paper discusses the technical challenges and details of introducing coverage-guided fuzzing and its associated binary instrumentation to Linux-based IoT platforms. The authors showcase AFL-IoT's strong performance and crash-finding ability through evaluations on benchmarks and real-world software.

Strengths

+ Authors attempt to discover and address challenges with IoT application fuzzing

Weaknesses

- Despite the authors' claims, the addressed challenges are not unique to fuzzing IoT devices and already addressed in previous work

- The core of the approach is the well-understood technique of elf/binary patching

- Insufficient evaluation

- Reliance on a simple fuzzer

Detailed comments for the author(s)

The paper needs significant editing for grammar and word use.

Use technical terms precisely how they are already used by the fuzzing community.

- e.g., "coverage-based fuzzing" should be "coverage-guided fuzzing"

- "track path information" -> do you mean "track branch coverage information"? AFL tracks branch coverage, not path coverage.

The paper claims that that IoT poses three unique challenges compared to conventional fuzzing: Lack of source code, limited resources, and network connectivity. Lack of source code (i.e., black-box) is common in existing fuzzing literature. Saying that limited resources is a novelty and then building an AFL-based prototype is non-sensical as AFL relies on simple mutation strategies. It would be much better to build a prototype based on concolic execution-based mutation. Lastly, one of the first uses for fuzzing was network facing devices, so this is not a unique challenge.

The heart of the approach is binary patching; an orthogonal and mostly solved problem.

In the Introduction you criticize Costin et al.'s reliance on inherently-inaccurate static analysis... yet, AFL-IoT also applies static analysis for identifying basic blocks. Is there a less hypocritical reason why their solution is insufficient?

10-14hr evaluations are too short; since publication of "Evaluating Fuzz Testing" the fuzzing community expects no less than 24hr evaluation trials. Additionally, any comparison of two or more fuzzers should present meaningful statistics like significance tests and effect sizes.

My main criticism of the instrumentation procedure is its offloading to a separate system. This detracts from AFL-IoT's scalability. Is it impossible or unreasonable to instrument the binary on the IoT device itself? You should discuss this design decision in detail.

The evaluation devices and binaries lack variety. Every device you use appears to be a network router, and all but one of the binaries (plugincenter) appear to be file compression utilities. Such little variety does not make for a compelling evaluation. Furthermore, the attack scenario isn't totally clear. How would an attacker interact with one of these devices in the real-world to exploit a memory corruption vulnerability? Can you demonstrate the real-world exploitability of crashes that AFL-IoT finds? Considering that all of the devices you listed are routers, I am having a tough time visualizing which attack vectors are open for attackers to provide malformed input to one of these vulnerable binaries.

As your results show, fuzzing with AFL-IoT is just as effective as fuzzing with regular AFL. But AFL is no longer the state-of-the-art fuzzer. Since your goal is to extend existing fuzzing techniques to a new platform, I think the paper would be a lot more substantive if you were to apply (or discuss applying) some more advanced fuzzing techniques (e.g., concolic/symbolic execution, taint tracking, etc.) as well. QSYM, Driller, and VUzzer are examples of approaches to look into.

The paper's focus on porting a Linux fuzzer to a Linux-based OS does not give it much depth. You say "most IoT devices are built upon RISC architecture"... What other non-RISC IoT's / non-Linux-based IoT OS's (e.g., FreeRTOS) exist? Can they be fuzzed? If so, how feasibly can they be fuzzed? How do they differ from x86 (the platform which the fuzzing community is most familiar with)? How would their respective architectures necessitate changes in AFL-IoT? Fuzzing these systems remains an open problem; this paper would offer a more significant contribution if it were to take a comprehensive look at fuzzing a multitude of IoT/embedded systems, and not just RISC Linux-based routers with file compression utilities.

Brief paper summary (2-3 sentences)

The paper proposes a lightweight coverage-based fuzzing framework for Linux-based IoT devices, called AFL-IoT. The AFL-IoT fuzzing technique has been designed to run natively on IoT devices. It addresses the core challenges of fuzzing on IoT devices, including the lack of source code, limited resources, and difficulty in interacting with the network daemon programs. The proposed approach leverages binary-level instrumentation to obtain program coverage information during fuzzing. Considering the limited resource on IoT devices, the binary instrumentation is performed on a powerful linux server. Then, the instrumented program along with the fuzzer is deployed on the IoT device for fuzzing. For coverage information, it leverages a static analyzer to identify all the basic blocks of the binary code and then injects binary code at the beginning of each basic block. New sections are inserted into the ELF binary file for the instrumented code and data. Toward interaction with network daemons, input redirection is performed by hooking socket APIs to map input from standard I/O stream to network sockets. The evaluation is performed on two benchmarks --one sample from t LAVA-M and several real-world IoT binaries.

Strengths

The main strength of the paper is the support for binary level instrumentation and the interaction with network daemon programs. The paper presents in-depth technical details on how to perform binary instrumentation in Linux binaries and patch executable (ELF) file. They also present details on how to handle corner cases, such as instrumenting instructions that use program counter values. In addition, the authors did a fair job in handling input redirection by hooking relevant APIs to map inputs obtained from fuzzers to the network sockets.

Weaknesses

-Incomplete evaluation

-Inadequate comparison with previous work

-Numerous inconsistencies in the evaluation as well as the technical presentation

Detail discussion of each of these follow

Detailed comments for the author(s)

Overall, the topic of the paper is interesting. While there has been a significant increase in the number of IoT devices recently, little attention has been given to security during their design phase, thus leaving numerous vulnerabilities in those devices. The authors should be applauded for their attempt to leverage fuzzing to find such vulnerabilities on IoT devices, which (as far as I can tell) is fairly new in this space. The proposed AFL-IoT seems reasonable and provides solutions to handle several corner cases.

That being said, the paper has numerous issues in the evaluation which makes it unsuitable for publication in its current state. Specific weaknesses include:

(1) The comparison of the proposed AFL-IoT approach with AFL (Table II) has been done for only one program of LAVA-M (i.e., uniq). The remainder of the programs -- base64, who and md5sum were not chosen for this experiment, which is very odd. The appears to be not justification for why uniq was selected. Given that LAVA-M is viewed as one of a number of de-facto fuzzing benchmarks, all the programs in LAVA-M should be used.

(2) The proposed AFL-IoT approach uses binary-level instrumentation for coverage information, yet, it is compared to the source code level instrumentation of AFL (Table II, Fig. 7). While I understand that the authors chose source code method of AFL, because AFL does not have similar binary instrumentation technique as the proposed AFL-IoT, it would be much relevant to add comparison with Qemu-based runtime instrumentation, which doesn't require source code.

(3a) The experiment was conducted for only 10 hours. As per the previous study [Klees, 2018], the fuzzing should be performed for a minimum of 24 hours for a fair comparison. A better analysis would be to show the performance over time, such as number of crashes obtained by each fuzzers for every few hours, for a minimum of 24 hours. The authors are encouraged to study [Klees 2018] when revising their work.

(3b) Table II compares proposed AFL-IoT with the state of art AFL fuzzer in terms of the number of unique crashes obtained for the “uniq” program of LAVA-M. First, it has not been explained whether these crashes are due to unique bugs. Recent study [Klees, 2018] demonstrates that even though AFL describes the crashes as unique, they may occur due to same bug. Second, even if these crashes are due to unique bugs, the number of bugs found by fuzzers in Table II are substantially low as compared to actual no. of unique bugs in “uniq” program. There are at least 28 unique bugs in the uniq program [Dolan-Gavitt, 2016], while Table II shows a maximum of only 6 bugs. What’s going on here?

(3c) There is a noticeable inconsistency in the evaluation. While, the correctness of AFL-IoT has been evaluated on only one program of LAVA-M, the performance evaluation has been done on all 4 programs of LAVA-M. Additionally, the compilation step in the performance evaluation has a different setting than the correctness evaluation. During correctness evaluation, afl-clang-fast was used, while for performance evaluation afl-clang was used. These difference in selection of compilation techniques can significantly vary the result and lead to a different conclusion. Therefore, for a fair and transparent evaluation, there must be consistency in selection of parameters and those must be chosen with proper justification.

(4) For a performance evaluation, to minimize randomness in fuzzing the experiment is repeated only for 4 times. On top of that, the fuzzers are just run for one hour on a single core, which is quite small amount of time for a fair comparison. One must follow standard practice and run the experiment for sufficient amount of time. As above, the experiment must show the execution speed must be measured over the time, such as every few hours for a minimum of 24 hours.

(5) The comparison with AFL is done only on LAVA-M benchmark, but not on any real-world binaries. Recent study [Klees, 2018] shows that even though many fuzzers perform fairly on LAVA-M, their performance vary significantly on the real-world programs. It is imperative that the comparisons be done based on some real-world binaries.

(6) How are the crashes classified as unique? In Sec. V-A (page 10), the authors state that “First, the number of unique crashes indicates the identified bugs”. This is not true because many crashes may be triggered due to same bug [Klees, 2018].

(7) Please describe specifically how the seed inputs were chosen. A clear explanation here is important because a fuzzer’s performance on the same program can be very different depending on what seed is used; even a valid but different seeds can induce very different behavior [Klees, 2018]

Suggestions for improvement beyond the above:

-It is difficult to understand the degree to which the majority of corner cases have been adequately dealt with. For that, it is important to provide more details regarding the 100 real-world binary programs that were fuzzed with AFL-IoT, such as binary size, their types (command line or network daemon), the device they were tested on (similar to Table III). Doing so will shed deeper insights on the diversity of binaries that were tested with the proposed fuzzer, and help understand it's applicability for fuzzing other binaries.

-The Wrap Solution (pg 6) needs proper explanation. This can be done by adding illustrations, accompanied by pseudocode or algorithms. Similarly, in page 8 (Sec. III-D) - fuzzing on network daemon can be better explained with proper illustrations. The discussion there is confusing.

-It is imperative that all the parameters that were used during compilation with afl-clang-fast be provided.

References:

[Klees, 2018] Klees, George, et al. "Evaluating Fuzz Testing." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018.

[Dolan-Gavitt, 2016] Dolan-Gavitt, Brendan, et al. "Lava: Large-scale automated vulnerability addition." Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 2016.

Brief paper summary (2-3 sentences)

This paper describes a technique which fuzzes embedded applications (traditionally a difficult target for fuzzing) by using binary rewriting to insert coverage instrumentation compatible with AFL and then fuzzing the target directly on the embedded device. The experiments test the bug-finding performance of AFL-IoT with one program from LAVA-M,

Strengths

+ Targets a class of programs that is currently very difficult to test

+ Reliable static binary rewriting for ARM is useful on its own

Weaknesses

- Targets Linux-based devices only

- Manual work still needed to get arbitrary code running on the device for fuzzing

- Unclear that fuzzing on the physical device is going to be faster or easier than QEMU (especially at scale)

- Experiments are mostly on open-source targets, for which AFL-IoT is not needed (the same testing could be performed more efficiently by recompiling on x86)

Detailed comments for the author(s)

The biggest issue I had with this paper is the question of whether, after all the work the authors did, it's actually preferable to simpler existing solutions. Specifically: it seems like one could simply run the target software in the existing AFL-QEMU, with some LD\_PRELOAD tricks to stub out problematic functions calls (similar to what Firmadyne did with libnvram: https://github.com/firmadyne/libnvram). Although the latter does require a bit of manual effort per target, it has the benefit of being able to run on commodity x86 hardware, and can be scaled up to as many parallel instances as you like. Fuzzing on an actual embedded device, by contrast, is limited by the number of physical devices you have.

There is also a question of novelty. The instrumentation for ARM code described sounds a lot like the technique used by Detours, which already exists and supports ARM code. Granted, Detours does not do static binary instrumentation by default (it injects instrumentation once a program is already running), but the amount of extra work needed to make it work statically is fairly small (as the authors note, ARM is simpler to rewrite than x86). Likewise, hooking network-related functions for fuzzing is a common technique, with existing implementations like preeny's desock (https://github.com/zardus/preeny).

The authors somewhat gloss over some of the practical issues involved in getting AFL-IoT to work on real targets. They mention that most embedded devices don't allow shell access by default in production, and say that they "use several tricks to activate these debug interface in order to enable shell access" – but these tricks are not detailed anywhere, and this could be nontrivial in the general case (consider how difficult it is to jailbreak an iPhone, for example).

The evaluation is also somewhat lacking. First, it's rather small for a modern fuzzing evaluation. Only one of the four LAVA-M programs is tested. The "real-world" fuzz testing evaluation only looks at a handful of busybox programs (on a Netgear R7000 and an ASUS AC1700), unzip (on an ASUS AC1700), and two closed-source utilities from a Xiaomi router.

Throughout the evaluation, how are unique crashes determined? For LAVA-M, ground truth about precisely which bug has been triggered is available. However, this is not available for other programs, and as Klees et al. ("Evaluating Fuzz Testing") demonstrate, simply using AFL's "unique crash" measurement can give highly inflated measures of the number of actual bugs discovered. On a similar note, for the bugs found in tsar and plugincenter – were these reported to Xiaomi, and are they new bugs? How many unique, previously undiscovered bugs were found by AFL-IoT?