Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing

We explored two techniques for identifying vulnerabilities in embedded software. First, we used a Large Language Model (LLM) to generate initial seeds for fuzz testing, enhancing efficiency. Second, we reused crash data from previous fuzzing sessions to streamline testing on similar binaries. These methods improved coverage and crash detection, making fuzz testing faster and more effective than traditional approaches.

Introduction

There has been a significant increase in the number of IoT (Internet of Things) devices, and so has been the increase in the number of cyber attacks on these devices. Embedded devices are the core part of the IoT ecosystem with firmware being the main logic governing most aspects of a system's behavior. Vulnerabilities in the firmware can indirectly impact the whole ecosystem. Firmware can be broadly classified into three categories: those based on modified generic operating systems (OS) like Linux, those based on real-time (RTOS) or custom operating systems, and those that do not have a formal operating system (non- OS or bare-metal). Each of these categories poses distinct challenges when it comes to security assessment[1] and often requires different approaches. To that end, we focus our attention in this work on the largest subclass of OS-based firmware: Embedded Linux.

As a critical set of commonly used utility programs in embedded Linux, BusyBox is a component of particular interest. It provides over 300 common Unix utilities within a single lightweight and compact executable, making it indispensable for resource-constrained Linux-based embedded devices. There are many IoT and OT (Operational Technology) devices running BusyBox, including remote terminal units (RTUs), human-machine interfaces (HMIs), and many others that are running on Linux. However, despite its many advantages, it can also present considerable risk, as it is often used with elevated privileges and provides multiple utilities that handle user input, which attackers have been able to exploit. 14 vulnerabilities were found in Busybox in 2021, some of which had the potential of remote code execution or denial of service attacks[2]. Despite this, in our investigation we have identified several real-world products that continue to use older versions of BusyBox that contain known vulnerabilities.

Fuzzing is a well-recognized software testing technique for uncovering vulnerabilities, but its effectiveness varies depending on the chosen target (i.e. firmware/software binary under test), each of which can present unique challenges. In this work, we propose and implement two techniques to assist software testing for embedded firmware. First, we leverage LLM-based

seed generation, in which we utilize commercial large language models (LLMs) to generate the initial input seeds for mutation-based, coverage-guided fuzzing. In doing so, we take advantage of LLMs' inherent capability to generate high-quality structured inputs that adhere to the input grammar of a target. Second, we employ a crash reuse strategy to identify crashes across similar binaries present in different embedded devices. This strategy is based on the intuition that an input that triggers a crashing vulnerability on one variant of a program is likely to trigger a crash on a different variant. This allows us to more efficiently determine if the same vulnerability is present on multiple program variants without performing fuzzing, thus saving significant time. When we mention a variant of a software component or similar binaries, we are referring to identical software components with varying version numbers or architectures or compiler optimization, or any custom modification by developers.

This research was done in collaboration with NetRise's [3] firmware security division. We sourced BusyBox ELFs from real-world embedded products collected from the company's proprietary firmware dataset, which had been collected using in-house extraction tools. Figure 1 shows the overall pipeline of the proposed work where we collect crashes using LLM-based seed generation technique, and then reuse the collected crashes to test the new variant of a similar target.

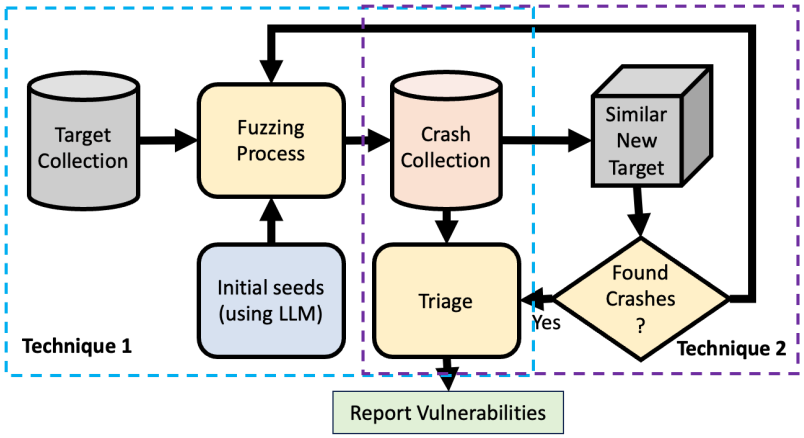


Figure 1 : Proposed work pipeline

Leveraging LLM for initial seed generation

We evaluated the first technique, LLM-based seed generation, by comparing control runs that used randomly generated initial seeds, to experimental runs that used initial seeds generated using OpenAI's GPT-4 LLM API. We observed a significant increase in crashes obtained when using LLM-generated seeds, demonstrating the potential for improving vulnerability detection. As a proof-of-concept, we demonstrate these techniques with AFL++ on BusyBox applets 'awk', 'dc', 'man', and 'ash'.

There are 2 scenarios under which initial seeds need to be generated: when the target input format is well-defined and/or standardized, and when the input format is loosely-defined or unknown. When the input format is well-defined, as would be the case for well-known programs like some BusyBox applets, we reason that LLM should not require additional training, as it already possesses knowledge of the expected input format through its initial training on the internet. This can also be determined empirically. However, when the input format of the target is ill-defined or unknown, as would be the case for custom communication protocols, LLM would require fine-tuning. In this scenario, LLM needs to be initially trained with known samples to develop an understanding of the expected input format. In the case of the BusyBox applets, we reason that GPT-4 should already be aware of the input format given the applet's popularity. Hence, we did not apply fine-tuning. We provided the following prompt to guide the seed generation process for awk:

*"**role**": "system", "**content**": "You are the initial seed generator for a fuzzer that has to fuzz BusyBox awk applet. In response only provide the list of awk scripts"*
*"**role**": "user", "**content**": "Generate initial seed to fuzz BusyBox awk applet"*

```
`BEGIN { FS=":"; OFS=":" } { print $1, $6 }`
`BEGIN { print "Start" } { print $1 } END { print "End" }`
`{ if ($1 > max) max = $1 } END { print max }`
`NR % 2 == 0`
`BEGIN {for(i=1;i<=3;i++) print "i=",i}`
`BEGIN { a["fruit"] = "banana"; a["number"] = 5; for (i in a) print i }`
`{ sum += $1 } END { print sum }`
`BEGIN { FS=" "; OFS="|"} { print $1, $3, $4 }`
`{ if ($1 > 10) print $0 }`
`BEGIN { OFS = "\n" } { print $1, $2 }`
`{ for (i = 1; i <= NF; i++) print $i }`
`BEGIN { a="Hello, World!"; print a }`
`{ if(NR > 1 && $1 > 1) print $0 }`
`BEGIN {print toupper("awk")} END {print tolower("AWK")}`
`{ if ($1 ~ /pattern/) print $0 }`
`{gsub(/match/, "replace"); print}`
```

Figure 2 : Generated list of commands w.r.t BusyBox 'awk' applet using LLM

The model responded with a list of commands relevant to the BusyBox awk applet as shown in Figure 2. These commands were then saved into individual .awk scripts, which were subsequently integrated into the input corpus. The generated corpus via LLM served as the set of initial seeds for the fuzzing process. We used afl-cmin to minimize the input corpus before sending it to the fuzzer, which filters the LLM-generated input corpus to include only the seeds that are useful for fuzzing.

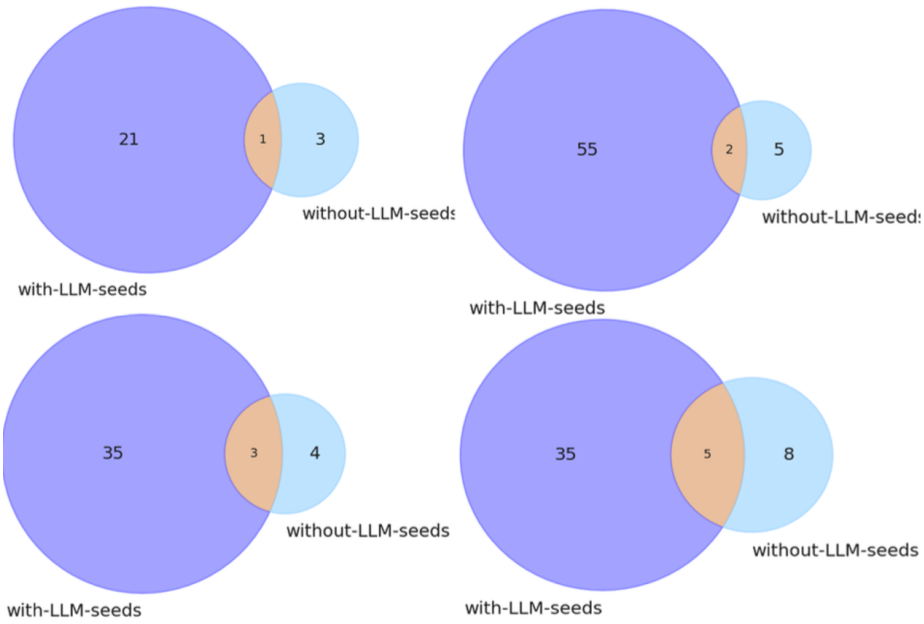


Figure 3 : No. of unique crashes with vs without LLM based initial input seeds. Target : Network controller, Network switch, storage array controller, firewall

We observed that more crashes were identified during fuzzing when the initial seeds were generated by LLM compared to the ones with random seeds. Moreover, a similar pattern was observed in the case of the number of edges found during the fuzzing of each of the targets. Refer to our paper[4] for more insights about the result. Figure 3 shows a Venn diagram depicting the number of unique crashes found in each case and the number of crashes common in both cases. It emphasizes the importance of discovering a more significant number of crashes. When there are more crashes to work with, there are more opportunities to discover different failing execution paths, thereby increasing the likelihood of uncovering vulnerabilities. It underscores this by revealing that more unique crashes were identified when utilizing LLM-generated initial seeds. AFL-Triage[5] was used to identify the unique crashes.

Crash Reuse

Having completed our fuzz testing runs on individual BusyBox targets (i.e. on different versions/variants of BusyBox binaries), we turn our attention to triaging crashes and investigating the potential utility of crash reuse. To recap, we have hypothesized that we can leverage known crashing inputs for a given target to quickly determine if variants of that target contain a similar vulnerability or bug. Crash Reuse provides advantages including efficiency, and performing blackbox testing. Initially testing the new variant of the target against the consolidated crash database offers the potential for significant time and resource savings. By capitalizing on the crashes identified during previous fuzz testing on similar targets, we can

leverage the resources previously expended in fuzzing and accelerate the fuzzer's coverage exploration by including it in future seeds. Hence, we can potentially identify previously discovered crashes in the new variant without extensive fuzz testing. Moreover, This technique is highly beneficial when conducting blackbox testing on new variants of a previously tested target. It is particularly advantageous in scenarios where the target utilizes accessible or open-source software components, even if further details are unavailable. By fuzzing open-source variants, we can gather crashing inputs to use as high-quality seeds that are likely to identify duplicate vulnerabilities. This is preferable to engaging in resource-intensive binary-only black-box fuzzing, which can be extremely difficult depending on the complexity of the system under test.

We actively curate a database of crashes obtained from previously fuzzed software components. Then, when we encounter variants of these software components in the future, we leverage the collected crashes to identify potential issues in the new variant under test without fuzzing. This provides us with a rapid initial assessment of the new target, which can later undergo more thorough fuzzing for in-depth inspection. This technique is applicable to any target whose variant has undergone previous fuzzing, and for which we possess a corresponding collection of crashes. After we had amassed a substantial number of crashes from fuzzed BusyBox targets, our total collection amounted to 4540 crashes that likely map to a much smaller place in the binary where the crash happens. Subsequently, we subjected the latest BusyBox version (v1.36.1 at the time of experiment) to testing against all these pre-existing crashes. This endeavor discovered 97 crashes in the latest BusyBox, of which 19 were unique. Later, we conducted traditional fuzzing on the latest BusyBox using AFL++ QEMU mode, with initial seeds generated by LLM over 10 hours. This approach yielded 20 crashes, of which eight were unique. Remarkably, five of these eight unique crashes were also identified using the crash reuse technique. Figure 4 presents a graphical comparison of the number of unique crashes discovered using the crash reuse technique versus traditional fuzzing, as well as the common crashes between the two methods. These results underscore the potential utility of crash reuse in software testing. Hence, it can reduce substantial time and resource demands, and is a valuable tool for black box fuzzing when a comprehensive crash database is available.

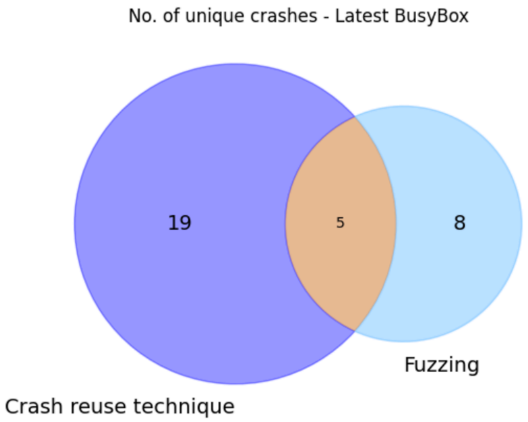


Figure 4 : Comparison of number of unique crashes in the latest BusyBox using crash reuse technique vs 10-hours of fuzzing.

Additionally, it is essential to note that not all crashes indicate software bugs. Crashes can occur for various reasons, including invalid inputs, false positives, unreachable code, execution environment factors, platform-specific issues, and other non-bug-related causes. Reaching conclusive determinations often involves meticulous manual triaging, which can be time-consuming and intricate. As such, we limited our scope to identifying crashes, with triaging performed only on a subset of crashes found in the latest BusyBox version, more detail about crash analysis can be found in the paper [4]. However, as previously discussed, the quantity of crashes is a vital metric in fuzzing. A higher number of unique crashes equates to a more extensive array of test scenarios to explore during testing. Consequently, this increases the likelihood of identifying potential software vulnerabilities or bugs.

Applicability of proposed techniques

The LLM-based technique for generating initial seeds plays a supportive role in the fuzzing process, contributing quality and diverse seeds that enhance fuzzing performance. However, it is crucial to note that this technique alone is not the sole factor influencing the overall outcome. Various associated factors, contingent upon the specific target, must be considered. The effectiveness is contingent upon the target type and the extent to which the initially provided diverse inputs contribute to code coverage. The primary function of LLM in this proposed technique is to assist in producing high-quality and diversified initial seeds, thereby potentially enhancing fuzzing performance. The seed generation using LLM requires initial manual intervention to validate if the generated seeds align with the target's requirements. In the case of a new target initially unknown to LLM, model training is essential for the target-specific seed format. However, this represents an initial, one-time effort; once the model learns the required seed format, it expedites the generation of diverse seeds suitable as potential initial seeds for fuzzing. Thus, we can leverage the knowledge base of LLM models, or train these models according to different target requirements. Therefore, this technique is not restricted to the BusyBox and can be adapted for use with different targets.

Similarly, the crash reuse technique proposed can be extended to various software components across different targets. The technique isn't limited to a particular target but applies universally. It can be employed in any scenario where we have previously gathered crash data by fuzzing a target and aim to test the variant of that target by reusing those crashes. The primary goal is to convey that the crash reuse technique could be beneficial for the initial screening of a new variant of a software component without spending hours on fuzzing. However, this technique may not uncover all the vulnerabilities requiring fuzzing for a thorough analysis.

Conclusion

In our pursuit to enhance existing software testing methodologies, we emphasize the significance of our proposed techniques, particularly within the context of embedded systems. Firmware in embedded systems often consists of numerous third-party software components with custom implementations and unique input types, making it predominantly a black-box testing scenario. The techniques introduced in this work, namely leveraging LLM for initial seed

generation and crash reuse, have exhibited promising outcomes that can significantly aid software testing efforts. While these techniques can be adapted for various targets, we have used them to analyze BusyBox for the proof of concept.

Nevertheless, it is essential to acknowledge certain limitations and challenges associated with these approaches. Utilizing LLM for initial seed generation may necessitate a significant initial effort, mainly when dealing with different targets, especially in the complex domain of embedded systems where a wide array of hardware protocols and custom input patterns are encountered. Furthermore, while the crash reuse technique represents a valuable first pass phase, it may not consistently identify all bugs, especially zero-day vulnerabilities. Hence, a traditional fuzzing technique remains a necessary complement for comprehensive testing. The crash reuse method primarily assists in determining whether previously identified crashes are applicable to a new target but does not guarantee the discovery of all potential bugs. However, while there are particular challenges and limitations, substantial research potential exists for harnessing these techniques to enhance and assist software testing endeavors. These approaches hold promise for improving the efficiency and effectiveness of testing procedures, particularly in the context of embedded systems and firmware analysis.

Acknowledgements

We would like to acknowledge the NetRise[3] team for providing us with a real-world embedded firmware database and cloud resources to perform a part of these experiments. We would also like to thank NSF CHEST for funding this project *(Project # 1916741 industry funding)*.

References

[1] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.
[2] Lucian Constantin. BusyBox flaws highlight need for consistent IoT updates, September 2021.
[3] netrise.io. Netrise | firmware security.
[4] Asmita, Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, and Houman Homayoun. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing. Usenix Security 2024 - Link
[5] Grant Hernandez. AFLTriage.