Debugger-driven Embedded Fuzzing

Max Eisele
Robert Bosch GmbH
Renningen, Germany
MaxCamillo.Eisele@de.bosch.com

***Abstract*—Embedded Systems – the hidden computers in our lives – are deployed in the billionths and are already in the focus of attackers. They pose security risks when not tested and maintained thoroughly. In recent years, fuzzing has become a promising technique for automated security testing of programs, which can generate tons of test inputs for a program. Fuzzing is hardly applied to embedded systems, because of their high diversity and closed character. During my research I want tackle that gap in fuzzing embedded systems – short: "Embedded Fuzzing". My goal is to obtain insights of the embedded system during execution, by using common debugging interfaces and hardware breakpoints to enable guided fuzzing in a generic and widely applicable way. Debugging interfaces and hardware breakpoints are available for most common microcontrollers, generating a potential industry impact. Preliminary results show that the approach covers basic blocks faster than blackbox fuzzing. Additionally, it is source code agnostic and leaves the embedded firmware unaltered.**

***Index Terms*—embedded systems, security testing**

I. Introduction

Google's OSS-Fuzz project [1] revealed over 30.000 bugs in about 500 open-source projects by using coverage-guided fuzzing. The attractiveness of coverage-guided fuzzing comes from its low setup effort and its automated, unsupervised way of generating test inputs. Prerequisites are an input interface, a crash detection, and a coverage feedback mechanism. Based on a set of inputs, called corpus, new test inputs are generated iteratively and forwarded to the target program. When new code coverage is reached during the target's execution, the responsible test input is added to the corpus. A crash of the target program indicates a bug and the responsible test input is preserved.

Embedded fuzzing comes with additional hurdles, compared to user applications on general purpose operating systems. Due to the closed nature of embedded systems, it is difficult to detect crashes of the target device [2] and gathering code coverage feedback is challenging.

One way to compensate missing observability is to let the embedded software run in an emulator. The transparency of an emulator not only allows to detect faults of the execution, but also to use memory access sanitizers and generation of code coverage feedback. Notably emulation-based approaches are, for instance, *HALucinator* [3], which enables to re-host firmware code at the hardware (HW) abstraction layer for more reusability and *P^2IM* [4], which attaches the fuzzer directly to the whole HW address space, to let it learn peripheral behavior. However, re-hosting embedded software into an emulator can be related to huge amounts of manual work, can lack of fidelity, and is an open research problem since years [5], [6].

As a result, emulators for embedded systems are rarely available and techniques for fuzzing on the actual HW have been developed. *IoTFuzzer* [7] hooks into accompanying smart phone applications from Internet of Things (IoT) devices to send fuzz data to the target device. *ARM-AFL* [8] has been proposed for on-target fuzzing on *ARM*-based embedded systems. *Harzer Roller* [9] is a linker-based function trace and sanitization approach demonstrated for *ESP8266* microcontrollers. For *ESP32* microcontrollers a basic block coverage feedback mechanism, based on static instrumentation, has been proposed by Boersig *et al.* [10]. Oh *et al.* [11] use software breakpoints for a dynamic instrumentation on microcontroller code. However, the approaches lack of broad applicability and mostly require to modify the firmware of the embedded system.

II. Debugger-driven Fuzzing

From my observation, a GNU Debugger (GDB) remote interface [12] is available for most microcontrollers, serving as a generic way of gathering insights from the execution of the System under Test (SUT). The GDB interface can be used to interrupt the execution, detect the execution of fault handlers, and to access memory and registers.

I also observed that most microcontrollers have HW breakpoint and data watchpoint registers. These can interrupt the execution upon reaching a corresponding code location when a specific memory location is accessed. The debugging host is then notified of the interruption and may resume execution. HW break and watch points do not disturb the execution as long as they are not triggered. However, only a limited amount of them is available, as they are represented by a unique HW register each. The ARM Cortex-M4 core, for instance, is designed for up to 8 HW breakpoints and 4 data watchpoints [13].

Research Goals

The goal of this thesis is to enable easy applicable, and efficient embedded fuzzing. I propose the following approaches:

- Coverage-guided fuzzing using HW breakpoints.
- Concolic execution using a debugger.
- Recovering input specifications using HW watchpoints.

Hardware breakpoints: enable feedback collection from the execution of the SUT. Breakpoints are set to interesting code locations, using the control flow graph of the program for guidance, which can be derived from source code or recovered with binary analysis tools like GHIDRA [14]. However, since the amount of HW breakpoints is limited, they must be placed cleverly. Also, transitive knowledge about other, non-selected basic blocks should be leveraged. For instance, by using the dominator graph of the control flow graph, other basic blocks that must have been executed before the current one can be determined [15].

Assuming a control flow graph, I propose the following strategies to place breakpoints:

- *Random Neighbor:* Randomly select basic blocks, whose parent block was already reached.
- *Reachability Count:* Prefer basic blocks, that have more successive (reachable) blocks.
- *Page Rank:* Prefer basic blocks with a higher *Page Rank* [16] score.

Using HW breakpoints to generate coverage feedback introduces an inevitable overhead, through (re-)setting operations and handling interrupts. On the other hand, finding new code blocks or paths during fuzzing usually follows a logarithmic curve over time [17]. Most coverage is found more at the beginning of a fuzzing campaign and renders the breakpoint interrupt overhead negligible over time. Overall, the overhead is expected to be dominated by the reset time of available breakpoints. The following research questions arise:

RQ1: How well can a limited amount of HW breakpoints aid a meaningful coverage-guided fuzzing of embedded systems?

RQ2: How does the number of available breakpoints affect the fuzzing performance?

RQ3: What is the dominating overhead and how to reduce it?

RQ4: How well is the introduced overhead compensated by faster input space exploration?

Concolic Execution: is a partial symbolic execution of the target's code based on a concrete execution trace. Since the debugger can be used for single stepping through code and reading memory, it can generate all necessary information to enable concolic execution with ANGR [18]. Thereby new input is generated, which can cover previously unreached code. However, trace generation with single stepping and the constraint solving from the symbolic execution can take a substantial amount of time, which leads to the research question:

RQ5: How much improves debugger-driven concolic execution for unreached basic blocks the performance of the previously presented coverage-guided fuzzing method?

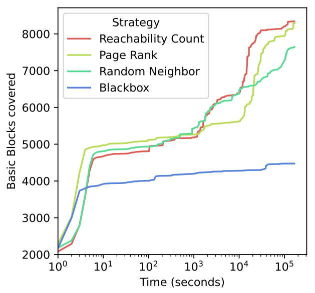
Recovered Input Specifications: can be used for efficient fuzzing without the need for coverage feedback. Gopinath *et al.* [19] showed that input grammars can be derived from a program by observing how and when the input buffer is accessed.

Data watchpoints can retrieve the same information by triggering an interrupt when the appropriate memory address is accessed. Therefore, the input buffer must be located within the address space and each test input must be executed several times, depending on its length and the amount of available HW data watchpoints. The related research question is:

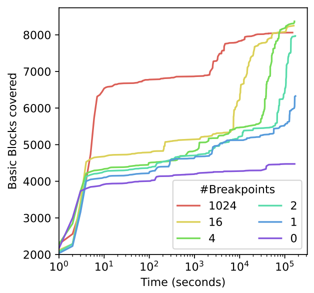
RQ6: How well can input specifications of an embedded system be recovered by using a limited amount of data watchpoints?

III. Preliminary Results

The actual fuzzer in the debugger-driven fuzzing method is interchangeable. In the current implementation, libfuzzer [20] is used. Following plots demonstrate the performance of the debugger-driven fuzzing method, compared to fuzzing without feedback. In order to simulate different amounts of available HW breakpoints, the results are obtained in an emulator.



(a) Comparison between different strategies, using 8 breakpoints.



(b) Comparison between different amounts of available breakpoints with the *PageRank* strategy.

Figure 1: Coverage over time plots, on fuzzing libjpeg-turbo [21] with debugger-driven fuzzing. Note that time axis is of logarithmic scale.

Figure 1a shows the coverage over time across the proposed strategies and blackbox fuzzing over 48 hours. Each breakpoint strategy outperforms blackbox fuzzing after a short while. Figure 1b shows how the number of available breakpoints influences the found coverage over time. As expected, a higher amount of breakpoints leads to a faster exploration of code.

IV. Evaluation and Expected Contribution

To show the generality of Debugger-driven Fuzzing I will first evaluate it on targets of the *Fuzzer Test Suite* [22] and thereby compare it against other software security testing tools, including blackbox fuzzing and emulation-based coverage-guided fuzzing. The latter represents an upper bound and is not expected to be exceeded. To demonstrate the broad applicability, I plan an evaluation on real products, in which case it is unlikely to get emulation-based coverage-guided fuzzing up and running.

With the proposed method, I expect to lower prerequisites and setup efforts for efficient embedded fuzzing. Debugger-driven embedded fuzzing could pose an easy-to-use, source code agnostic, and non-invasive method for security testing of embedded systems.

Acknowledgment

My thesis is supervised by Prof. Andreas Zeller from CISPA Helmholtz Center for Information Security, Saarland and I carry out this work as a PhD candidate at Saarland University.

References

[1] Google, "OSS-Fuzz," https://google.github.io/oss-fuzz/, 2021, accessed: 2021-12-20.

[2] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.

[3] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1201–1218.

[4] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1237–1254.

[5] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, "SoK: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 687–701.

[6] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–36, 2021.

[7] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[8] R. Fan, J. Pan, and S. Huang, "ARM-AFL: Coverage-guided fuzzing framework for ARM-Based IoT devices," in *International Conference on Applied Cryptography and Network Security*. Springer, 2020, pp. 239–254.

[9] K. Bogad and M. Huber, "Harzer roller: Linker-based instrumentation for enhanced embedded security testing," in *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, 2019, pp. 1–9.

[10] M. Börsig, S. Nitzsche, M. Eisele, R. Gröll, J. Becker, and I. Baumgart, "Fuzzing framework for ESP32 microcontrollers," in *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, 2020, pp. 1–6.

[11] J. Oh, S. Kim, E. Jeong, and S.-M. Moon, "OS-less dynamic binary instrumentation for embedded firmware," in *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. IEEE, 2015, pp. 1–3.

[12] B. Gatliff, "Embedding with gnu: the gdb remote serial protocol," *Embedded Systems Programming*, vol. 12, pp. 108–113, 1999.

[13] Arm, "Arm Cortex-M4 Technical Reference Manual," https://developer.arm.com/documentation/100166/0001, 2009, accessed: 2022-01-03.

[14] National Security Agency, "Ghidra," https://ghidra-sre.org/, 2019, accessed: 2021-12-20.

[15] H. Agrawal, "Efficient coverage testing using global dominator graphs," in *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 1999, pp. 11–20.

[16] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[17] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 713–724.

[18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[19] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 172–183.

[20] LLVM, "libfuzzer a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, accessed: 2021-11-22.

[21] "libjpeg-turbo," https://libjpeg-turbo.org/, accessed: 2021-11-22.

[22] "Google fuzzer test suite," https://github.com/google/fuzzer-test-suite, accessed: 2021-11-22.