# E-FuzzEdge: Efficient In-Place Firmware Fuzzing via Parallel Scheduling (Short Paper)

Davide Rusconi*, Osama Yousef*, Mirco Picca*, Danilo Bruschi*, Flavio Toffalini†, and Andrea Lanzi*
*University of Milan, †Ruhr-Universität Bochum
davide.rusconi@unimi.it, ossama.yousef@studenti.unimi.it, mirco.picca@unimi.it, danilo.bruschi@unimi.it,
flavio.toffalini@rub.de, andrea.lanzi@unimi.it

***Abstract*—In this paper, we show E-FuzzEdge, a novel fuzzing architecture targeted towards improving the throughput of fuzzing campaigns in contexts where scalability is unavailable. E-FuzzEdge addresses the inefficiencies of hardware-in-the-loop fuzzing for microcontrollers by optimizing execution speed. We evaluated our system against both real-world embedded libraries and state-of-the-art benchmarks, demonstrating significant performance improvements. A key advantage of the E-FuzzEdge architecture is its compatibility with other embedded fuzzing techniques that perform on device testing instead of firmware emulation. This means that the broader embedded fuzzing community can integrate E-FuzzEdge into their workflows to enhance overall testing efficiency.**

## I. Introduction

Embedded devices are ubiquitous in safety-critical domains such as transportation, healthcare, and industrial automation. While resource constraints demand optimized time and space efficiency, security remains paramount. The rising number of cyberattacks on IoT systems underscores the urgent need for effective vulnerability testing. However, applying traditional analysis tools to embedded systems is challenging due to their unique architecture: severe resource limitations, a lack of filesystems or processes, and often minimal or no operating systems.

Recent efforts have adapted fuzzing—one of the most effective vulnerability detection techniques—to embedded systems through two main approaches: *Emulation-based techniques* execute firmware in virtualized environments, offering scalability and introspection but relying on accurate peripheral modeling, which is difficult given proprietary or undocumented hardware. *In-place fuzzing* executes tests directly on physical devices, ensuring realistic interaction with actual peripherals and reducing false positives; however, it suffers from scalability challenges and communication overhead. Existing systems like IPEA [1] and $\mu$AFL [2] exemplify hardware-in-the-loop approaches, using minimalistic instrumentation and ARM debugging capabilities, respectively.

In this paper, we introduce *E-FuzzEdge*, an in-place fuzzer that enhances execution efficiency through architectural optimizations that are orthogonal to existing approaches. Our key insight is that the communication overhead between the fuzzing host and the embedded device is the primary bottleneck in hardware-in-the-loop fuzzing. E-FuzzEdge addresses this through two mechanisms: (1) minimizing data transfer by performing coverage analysis on-device and transmitting only compact feedback, and (2) enabling parallel fuzzing instances on the host to eliminate device idle time during feedback processing.

We implemented a prototype based on AFL++ and evaluated it against IPEA and $\mu$AFL. Our results demonstrate significant throughput improvements—up to 2× with two parallel processors on STM32L0 hardware. Importantly, our architectural contributions are orthogonal to existing techniques, meaning they can be integrated into other embedded fuzzers to further enhance performance while preserving their unique innovations.

## II. Background

### A. Fuzz Testing

Fuzzing is an automated testing technique that generates test inputs and monitors program execution for unexpected behaviors [3]. Modern fuzzers [4] consist of key components: an executor that runs the target program, observers that collect coverage data, a feedback system that identifies interesting inputs, mutators that generate new test cases, and a scheduler that selects inputs from the corpus.

Fuzzers are classified by input generation (*generational* vs. *mutational*), program knowledge (*black-box*, *white-box*, or *grey-box*), and execution model. Gray-box fuzzers like AFL++[5] use lightweight instrumentation for coverage feedback, balancing effectiveness and efficiency. AFL++ employs a *forkserver* model, spawning a new process per test case for isolation; however, this incurs overhead. *In-process fuzzing* improves speed by sharing address space but risks state persistence across executions.

### B. Fuzzing in Embedded Systems

Embedded devices face unique constraints—limited memory, processing power, and often no filesystem or operating system—making traditional fuzzing impractical. Two main approaches have emerged: *Emulation-based fuzzing* [6], [7], [8] virtualizes firmware for scalable testing but depends on accurate peripheral modeling, which is challenging for proprietary

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23004
www.ndss-symposium.org

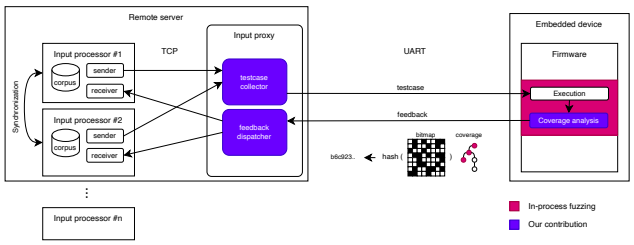


Fig. 1. E-FuzzEdge Architecture

hardware. *In-place fuzzing* [2], [1] tests directly on physical devices, ensuring realistic peripheral interaction and reducing false results; however, it suffers from scalability issues due to resource constraints and communication bottlenecks between the fuzzer and the device.

## III. E-FuzzEdge Design

The main focus of our project is designing an efficient greybox in-place fuzzing architecture for embedded devices, minimizing execution overhead while maintaining adaptability across different hardware environments. The core requirements of our architecture are versatility and efficiency. We can examine the former along two dimensions: the adaptability of the communication channel to suit the diverse environments typical of embedded devices and a decrease in platform requirements, such as processes and filesystems. This adjustment is necessary for platforms with extremely simplistic or non-existent operating systems. In terms of efficiency, the main issue is the additional burden caused by the communication channel. As this burden is intrinsic to the platform, our method emphasizes reducing its impact instead of trying to remove it altogether. We address this overhead by employing two main strategies: decreasing the volume of data transmitted through the communication channel and reducing device idle time through enhanced data transmission and processing.

### A. Architectural Model

We devised our architecture to satisfy the mentioned requirements, particularly the one related to overhead, since versatility pertains to the implementation and not the model. Our system remains consistent with standard forkserver-based fuzzing designs, but we decouple input processing,mutation, and scheduling from test case execution and feedback collection. Figure 1 shows the three elements upon which our system is built: multiple *input processors*, an *input proxy*, and a single *input executor*

*a) Input Processor:* Multiple processors run on a desktop device, leveraging superior computing resources and filesystem access to enhance usability and accelerate operations that do not require direct interaction with the embedded hardware. They are tasked with all operations related to processing test cases, such as scheduling and mutation, based on coverage feedback. The choice to use multiple *input processors* stems from the idea of maximizing the utilization of the device. As previously mentioned, sending and retrieving data on the communication channel is a slow process; the use of a one-to-many architecture ensures that the single executor does not remain idle while a processor is retrieving and analyzing the data but rather runs the inputs provided by the other executors, thus maximizing throughput.

*b) Input proxy:* A lightweight middleware sits between processors and devices. It exposes a stable host-facing interface and adapts to the device link (TCP or UART). For UART, it handles practical details like DMA cyclic reception and padding, so half/full-buffer interrupts fire predictably. The choice of a proxy comes with the benefit of easing the process of changing the communication protocol between the desktop system and the embedded device, which is particularly relevant in a highly heterogeneous context regarding communication protocols.

*c) Input Executor:* The executor runs in-process with the target firmware and a small harness without relying on processes or fork(), which may not be available when fuzzing bare-metal. The harness is tasked with initializing the necessary peripherals for testing the firmware, retrieving inputs from the proxy, and feeding them to the program when needed. The execution of test cases follows the standard fuzzing loop of execution and feedback analysis; however, it is a persistent process, meaning that the system is not rebooted after each test case. While this approach can create issues due to separate tests coinciding in the stimulation of a bug. We argue that this issue can be mitigated by careful harness design, and the benefit in terms of executions per second makes adopting the persistent approach worth it. The executor is also tasked with processing coverage data related to the executed test; this allows us to minimize the data transmitted over the communication channel in the form of a flag indicating whether new bits were found and a checksum, rather than performing a full-bitmap transmission that would significantly hinder fuzzing performance.

### B. Communication Protocol

The protocol mirrors the classic fuzzing loop of scheduling, mutating, executing, and evaluating, but it is engineered to minimize bytes in flight and device idle time. During initialization, processors establish TCP sessions with the proxy, which opens or attaches to the device link, and the executor configures its peripherals and allocates the coverage structures it will maintain during the campaign. Any static configuration that affects message formats, such as the effective map size or maximum payload, is broadcast from the proxy to processors so that formatting remains consistent.

In a steady state, each processor prepares a test case and submits it to the proxy; the proxy forwards the frame to the device; the executor runs the harnessed target to completion or timeout, computes coverage locally, and performs on-device triage against the cumulative map. It then sends a compact feedback record composed of a fault code, when applicable, a boolean flag indicating whether new bits were observed, and, if enabled, a short checksum. The proxy routes this record back to the originating processor, which decides whether to

retain or discard the input and immediately schedules the next one. Because only minimal feedback travels on the hot path, steady-state traffic remains small and predictable.

Transport specifics are handled transparently by the proxy. Over UART, the executor configures DMA in cyclic mode and relies on half- and full-buffer interrupts to delimit frames, while the proxy pads headers and bodies so those interrupts fire deterministically.

## IV. Implementation

In order to test our model, we developed a prototype based on AFL++ [5]. Although AFL++ already provides a proxy system, we developed the remote fuzzing system from scratch because this system allows only a one-to-one fuzzing architecture and requires the fuzzer to send back the entire coverage map, significantly slowing down the entire process. Additionally, we modified the fork server to allow remote communication with the embedded device and adapted the feedback collection mechanism to process the input without the need for the coverage map since the task of analyzing the input is performed by the input executor. Moreover, we modified the compiler runtime to accomplish two objectives: first, we reduced the data needed for the fuzzing campaign to meet the strict space requirements present within the embedded context; second, we added the infrastructure necessary to initialize the peripherals required for communicating with the proxy and retrieving input data from it. Lastly, we developed the proxy system to mediate between the afl instances and the embedded system. In total, we contributed 6534 lines of C/C++ code.

## V. Evaluation

We evaluate E-FuzzEdge in two settings: a desktop-only setup where input processors and the executor communicate over TCP, and a hardware-in-the-loop setup using an STM32L053R8 board with a desktop host (Intel i7-9750H, 16 GB RAM). In the embedded setup, host–device communication uses UART. For each firmware, we ran seven 4-hour fuzzing campaigns for $n \in \{1, 2, 3, 4\}$ parallel input processors, reporting executions per second (exec/s) aggregated over each campaign.

### A. Dataset

We used `xpdf` for desktop evaluation. For embedded testing, we used 10 open-source firmwares: two from the OAT dataset [9] and eight from Arduino Projects [10] and Raspberry Pi Pico Examples [11]. Each firmware was modified to replace `main` with a simple in-process harness (standard fuzzing practice), and input functions (`fgetc`, `scanf`) were redirected to read from proxy-provided test cases rather than `stdin` or UART.

### B. Results

*a) Desktop Setting:* To evaluate our host-side concurrency model under optimal conditions, we tested the protocol over TCP with a single forkserver-backed executor.

TABLE I
Number of executions per second for each firmware.

| Firmware name | 1 Instance | 2 Instances | 3 Instances | 4 Instances |
|---|---|---|---|---|
| discokeyboard | 14.36 (1.00x) | 21.17 (1.47x) | 10.03 (0.70x) | 10.42 (0.73x) |
| huemotion | 14.31 (1.00x) | 21.68 (1.52x) | 10.03 (0.70x) | 10.08 (0.70x) |
| ledmatrixpainter | 18.59 (1.00x) | 23.71 (1.28x) | 23.93 (1.29x) | 12.01 (0.65x) |
| miniigstats | 7.41 (1.00x) | 4.64 (0.63x) | 4.67 (0.63x) | 4.60 (0.62x) |
| modbus | 16.79 (1.00x) | 25.76 (1.53x) | 26.07 (1.55x) | 27.89 (1.66x) |
| musiccontroller | 0.94 (1.00x) | 0.94 (1.00x) | 0.95 (1.01x) | 0.94 (1.00x) |
| pixelpainter | 2.62 (1.00x) | 2.62 (1.00x) | 2.61 (1.00x) | 2.61 (1.00x) |
| rflock | 15.89 (1.00x) | 23.94 (1.51x) | 21.94 (1.38x) | 20.00 (1.26x) |
| thermostat | 16.62 (1.00x) | 32.96 (1.98x) | 33.32 (2.00x) | 24.70 (1.49x) |
| xml | 3.22 (1.00x) | 4.17 (1.30x) | - | - |
| Total | 1.00x | 1.27x | 1.06x | 0.95x |

Baseline throughput with one processor is **20.24 exec/s**. Multiple processors yield speedups of $2.09\times$, $2.10\times$, $3.20\times$, $5.11\times$, and $6.81\times$ for $\{2, 3, 4, 6, 8\}$ processors, reaching $\{42.28, 42.44, 64.69, 103.45, 137.96\}$ exec/s. Scaling becomes sub-linear as available cores saturate.

This stress test isolates the effect of low-latency transport and host-side concurrency from device constraints, approximating performance on high-end boards with fast links. While typical desktop fuzzers achieve thousands of exec/s without our proxy architecture, this experiment demonstrates two key insights: (1) when the link is fast, parallel processors directly increase throughput until the CPU becomes the bottleneck, and (2) the saturation point depends on per-input computation versus execution overhead. This mirrors our embedded results, where saturation occurs earlier due to slower links and device constraints.

*b) Embedded Setting:* Table I reports executions per second (exec/s) by firmware and the number of parallel processors. Values in parentheses show speedups relative to one processor; the "Total" row shows the geometric mean speedup across the firmware. The key result is that adding a *second* processor consistently improves throughput, with a geometric-mean speedup of **1.27**$\times$. A third processor yields $1.06\times$, and a fourth $0.95\times$, indicating diminishing returns beyond two. Specifically, 7/10 firmware improvements were observed with two processors (gains $\approx$1.3–2.0$\times$), 2/10 remained unchanged, and 1/10 regressed. This reflects the intended design: a second processor keeps the executor busy while the first processes feedback, but additional processors mainly add overhead once the device and link saturate.

Saturation is target-specific: optimal performance occurs at $n = 2$ for 4/10 firmwares (`discokeyboard`, `huemotion`, `rflock`, `xml`), at $n = 3$ for 3/10 (`ledmatrixpainter`, `thermostat`, `musiccontroller`), at $n = 4$ for `modbus`, and at $n = 1$ for `miniigstats` and `pixelpainter` (which have fixed delays). When channel overhead dominates, additional processors create contention; when targets require higher per-input computation (e.g., `modbus` protocol parsing), extra processors provide near-linear gains. For UART links and STM32L0-class boards, two processors are a strong default.

## VI. Discussion

Our evaluation has demonstrated the effectiveness of our parallel architecture; however, it has also highlighted several

key limitations and avenues for future work.

*a) Evaluation scope:* As shown in the previous section, E-FuzzEdge improves fuzzing throughput when multiple input processors handle test cases in parallel. Our evaluation, however, is limited to a single MCU family (STM32L0) and two channels (TCP on desktop, UART on the device). While we do not expect an inversion of the observed trend, more powerful boards and faster links will likely shift the saturation point beyond two processors. Replicating on additional MCUs and transports (e.g., USB CDC/SPI) is future work.

*b) Memory overhead:* Our prototype maintains one temporary coverage map per execution and one cumulative map per input-processor instance on the device. On memory-constrained boards with large targets, these allocations may be prohibitive. A straightforward extension is to use a single cumulative map shared across processors, reducing redundancy; this requires careful evaluation to rule out contention and other side effects.

*c) Number of input processors:* The current level of parallelism is fixed at the beginning of the campaign. Results indicate that the optimal number of processors depends on both firmware and target hardware. On the STM32L053R8, two processors are typically the best choices. We plan to improve in this area by enabling runtime negotiation of the number of input executors; this approach can dynamically increase or reduce the number of input processors to keep the board saturated while minimizing the overhead caused by the communication channel in order to optimize system throughput.

*d) Crash Handling:* Currently, we rely on timeouts to detect crashes. While this is a common and practical approach, it cannot distinguish a true crash from a simple timeout due to a delay in the target firmware's execution. This limitation can be addressed by implementing custom error handlers within the firmware to signal the host when a crash occurs. Furthermore, our architecture's high throughput, while beneficial, also exacerbates a core problem of in-place fuzzing: the need for frequent device resets upon crash detection. An increased crash discovery rate, while a sign of a successful fuzzing campaign, can lead to more frequent reboots, which, in turn, can significantly impact overall performance and limit the practical benefits of our architecture.

## VII. Related Work

Embedded fuzzing techniques fall into two categories: emulation-based and hardware-in-the-loop approaches.

**Emulation-Based Approaches:** These techniques execute firmware in simulated environments without physical hardware, enabling scalable testing. DICE [12], Fuzzware [6], P2IM [7], and Laelaps [13] use symbolic execution, peripheral abstraction, and model learning to approximate hardware peripheral behavior. Rehosting approaches like HALucinator [8] intercept hardware abstraction layer (HAL) calls with simulated responses, while Pretender [14] models Memory-Mapped I/O (MMIO) peripherals by learning from actual hardware behavior. Despite enabling scalability and deep introspection, emulation-based approaches face challenges with peripheral modeling accuracy, particularly for proprietary or undocumented hardware, which may lead to discrepancies in actual firmware behavior.

**Hardware-in-the-Loop Approaches:** These techniques execute firmware directly on target devices for more accurate testing. IPEAFuzz [1] uses lightweight instrumentation for efficient data exchange and feedback collection, while $\mu$AFL [2] employs ARM-specific debugging mechanisms (hardware breakpoints and trace features) for instrumentation-free fuzzing. HIL approaches provide high-fidelity testing with actual hardware but suffer from scalability limitations and hardware dependencies, constraining execution speed and reproducibility compared to emulation.

Our work addresses HIL scalability challenges through architectural optimizations that are orthogonal to existing techniques. By minimizing communication overhead and enabling host-side parallelism, E-FuzzEdge improves throughput while maintaining the fidelity advantages of hardware-in-the-loop testing.

## VIII. Conclusion

In this work, we presented E-FuzzEdge, an in-place fuzzer for embedded systems. We showed that parallelizing fuzzing instances, even with a single on-device executor, significantly improves throughput. This result is orthogonal to existing approaches and can directly benefit in-place embedded fuzzers such as IPEA Fuzz [1] and $\mu$AFL [2]. Finally, we implemented and released an AFL++-based prototype, bridging the gap between state-of-the-art desktop greybox fuzzing and current embedded fuzzing tools.

## Acknowledgment

This work was supported in part by the SERICS project (PE00000014) under the NRRP MUR program, funded by the European Union – NextGenerationEU (NGEU). The views expressed are those of the authors and do not necessarily reflect those of the European Union or the Italian Ministry of University and Research (MUR). This work was also supported by the Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

## References

[1] J. Shi, W. Li, W. Wang, and L. Guan, "Facilitating non-intrusive in-vivo firmware testing with stateless instrumentation," 01 2024.

[2] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "$\mu$afl: non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–12. [Online]. Available: https://doi.org/10.1145/3510003.3510208

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, Dec. 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[4] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602

[5] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[6] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[7] 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)*. USENIX Association, Aug. 2020, pp. 1237–1254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[8] 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)*. USENIX Association, Aug. 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[9] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1433–1449.

[10] "Arduino project," https://github.com/mattiasjahnke/arduino-projects.

[11] "Raspberry Pi pico-example," https://github.com/raspberrypi/pico-examples.

[12] A. Mera, B. Feng, L. Lu, and E. Kirda, "Dice: Automatic emulation of dma input channels for dynamic firmware analysis," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, ser. S&P/Oakland'21, May 2021.

[13] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 746–759. [Online]. Available: https://doi.org/10.1145/3427228.3427280

[14] E. Gustafson, M. Muench, C. Spensky, , N. Redini, A. Machiry, A. Francillon, D. Balzarotti, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," 2019.