# Toward Hybrid Rehosting for Efficient IoT Firmware Fuzzing

Jiawei Yan[1] Takahiro Shinagawa[1]

## Extended Abstract

Nowadays, attacks targeting IoT devices have become increasingly frequent. To identify vulnerabilities in IoT devices, fuzzing remains one of the most widely used techniques for vulnerability analysis. However, before performing fuzzing, it is essential to conduct firmware rehosting to enable high-performance fuzzing. Rehosting refers to the automated creation of virtual execution environments for embedded firmware, which involves recreating the behavior of one or more firmware services within an emulated environment. Generally, there are two types of rehosting techniques: System-Level Rehosting and Process-Level Rehosting. System-Level Rehosting provides full software-based emulation of an IoT device's entire hardware and software stack, including the operating system kernel, user-space processes, and hardware peripherals. This approach enables the complete firmware of an embedded device to be executed and analyzed in a virtualized environment that closely replicates the behavior of the physical device. In contrast, Process-Level Rehosting emulates an individual process within the context of the original filesystem, reconstructing only the essential components required for that process to run.

However, existing Process-Level Rehosting techniques face significant challenges. As mentioned earlier, to improve fuzzing efficiency compared to System-Level Rehosting, Process-Level Rehosting emulates only a single process within the emulated environment. When the emulated process needs to communicate with other processes or services, it cannot function correctly, leading to rehosting failure. For example, many IoT devices use the Linux-based OpenWrt project as their firmware development environment. In OpenWrt, a service called `ubus` provides system-level inter-process communication (IPC) between various daemons and applications. The `ubus` service is also essential for user applications to read and modify the configuration of IoT devices. Because the emulated process fails to communicate with the `ubus` service, it behaves abnormally, and Process-Level Rehosting ultimately fails.

[1] The University of Tokyo

Prior studies have proposed various firmware rehosting techniques to support dynamic analysis and fuzzing of IoT devices. System-Level Rehosting frameworks such as HALucinator [1] emulate entire hardware and software stacks by abstracting peripheral models, achieving high fidelity but suffering from substantial performance overhead. In contrast, Process-Level Rehosting approaches like FIRM-AFL [2] and EQUAFL [3] accelerate fuzzing by emulating individual processes extracted from firmware images, yet they fail when inter-process communication or shared services are required. Recent protocol-aware approaches such as that by Bley et al. [4] attempt to improve compatibility in System-Level Rehosting by modeling network stacks, but they do not address IPC dependencies between processes. Our work complements these studies by integrating Process-Level and System-Level Rehosting to enable efficient and faithful emulation of IPC-dependent IoT applications.

To address this problem, we propose an approach that integrates System-Level Rehosting into Process-Level Rehosting. By allowing the emulated process to communicate with the full system emulation environment, IPC requests from the emulated process to the `ubus` service can be correctly handled. On the Process-Level Rehosting side, we intercept all system calls related to `ubus` and forward them to the System-Level Rehosting side for execution. After the system emulator completes the forwarded system call, we transfer its output back to the Process-Level Rehosting environment, restore the CPU state, and resume the emulated process. To ensure the correctness of system call execution, we synchronize memory between the two emulation environments by sharing the process's page table.

Figure 1 illustrates the overall implementation structure. In system emulation, we launch the same program as in the process emulation. After its memory initialization, we pause this process and wait for system calls from the process emulation. In process emulation, we inject a custom library into the emulated process to hook the `libubus.so` library, which manages communication with the `ubus` service. This hook library marks all system calls originating from `libubus.so`. When such a system call is invoked, we transfer the CPU state to the system emula-

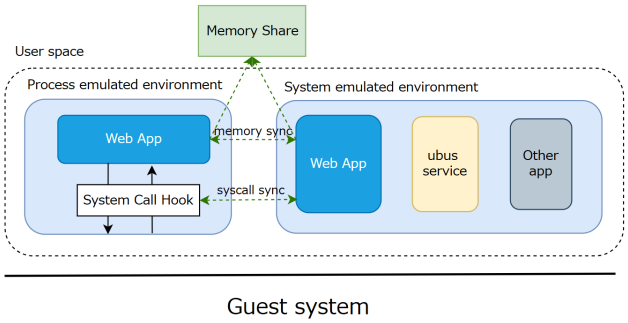


**Fig. 1** Overall design

tion and pause the process emulation until the call returns. After the system emulation completes the system call and returns from the kernel, we restore the CPU state to the process emulation and resume execution. Because context switches occur during system call execution, we must identify the correct return point. We record the value of the page table register (for example, CR3) of the target process and compare it with the current value upon return. Memory synchronization is also required to ensure correct system call execution. To avoid copying memory between the two processes, we synchronize page tables and let the system emulation handle all page faults. When a page fault occurs in the process emulation, we forward it to the system emulation for handling. After handling, we map the corresponding virtual memory to the associated physical memory in the process emulation.

Currently, our prototype successfully performs hybrid rehosting for simple processes in the OpenWrt environment. This demonstrates the feasibility of integrating System-Level and Process-Level Rehosting to enable IPC-dependent applications to run correctly. As future work, we plan to extend our implementation to support more complex processes that involve multiple system services, threads, and peripheral interactions. By improving synchronization mechanisms and extending system call forwarding coverage, we aim to achieve robust and scalable hybrid rehosting for a wider range of IoT firmware.

**Acknowledgments** This work was supported by JST, CREST Grant Number JPMJCR22M3, Japan.

## References

[1] Clements, A. et al.: HALucinator: Firmware Re-hosting Through Abstraction of Peripheral Models, *Proceedings of the 29th USENIX Security Symposium* (2020).

[2] Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H. and Sun, L.: FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation, *Proceedings of the 28th USENIX Security Symposium* (2019).

[3] Zheng, Y., Li, Y., Zhang, C., Zhu, H., Liu, Y. and Sun, L.: Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation, *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, p. 417–428 (online), DOI: 10.1145/3533767.3534414 (2022).

[4] Bley, M., Scharnowski, T., Wörner, S., Schloegel, M. and Holz, T.: Protocol-Aware Fimrware Rehosting for Effective Fuzzing of Embedded Network Stacks, *Proceedings of the 32nd ACM Conference on Computer and Communications Security*, (online), DOI: 10.1145/3719027.3765125 (2025).