SURGEON: Performant, Flexible, and Accurate Re-Hosting via Transplantation

Florian Hofhammer[1], Marcel Busch[1], Qinying Wang[1,2], Manuel Egele[3], Mathias Payer[1]
[1]EPFL, Switzerland. {firstname.lastname}@epfl.ch
[2]Zhejiang University, China.
[3]Boston University, USA. megele@bu.edu

***Abstract*— Dynamic analysis of microcontroller-based embedded firmware remains challenging. The general lack of source code availability for Commercial-off-the-shelf (COTS) firmware prevents powerful source-based instrumentation and prohibits compiling the firmware into an executable directly runnable by an analyst. Analyzing firmware binaries requires either acquisition and configuration of custom hardware, or configuration of extensive software stacks built around emulators. In both cases, dynamic analysis is limited in functionality by complex debugging and instrumentation interfaces and in performance by low execution speeds on Microcontroller Units (MCUs) and Instruction Set Architecture (ISA) translation overheads in emulators.**

SURGEON provides a performant, flexible, and accurate re-hosting approach for dynamic analysis of embedded firmware. We introduce *transplantation* to transform binary, embedded firmware into a Linux user space process executing natively on compatible high-performance systems through static binary rewriting. In addition to the achieved performance improvements, SURGEON scales horizontally through process instantiation and provides the flexibility to apply existing dynamic analysis tooling for user space processes without requiring adaptations to firmware-specific use cases. SURGEON's key use cases include debugging binary firmware with off-the-shelf tooling for user space processes and fuzz testing.

I. INTRODUCTION

MCU-based embedded systems are ubiquitous and enable a plethora of applications such as medical and health-related applications [25], [53], [58], [71], smart home and infrastructural use cases [40], or industrial production and control systems [19]. The lack of Memory Management Unit (MMU)-based memory isolation, the prevalence of memory-unsafe languages such as C/C++, as well as exposure of such devices to the Internet make them an attractive target for malicious actors and security researchers. These properties consequently call for thorough testing and analysis of such embedded firmware to mitigate the potentially fatal impact of vulnerabilities on the physical world.

Static analysis of software suffers from high false positive rates for vulnerability detection [29], [39], [43]. This approach becomes even more challenging when analyzing binary-only firmware where source-based information such as type information is lost during compilation. Hence, dynamic analysis techniques build a cornerstone of modern software analysis. Such techniques include fuzz testing, dynamic taint tracking, symbolic execution, or simple execution in a debugger to gather runtime information about the analyzed software.

However, COTS embedded devices commonly lack the debug interfaces required for introspection into the firmware's execution. Even if such interfaces are available, low-power MCUs come with limited on-device compute resources and restrict the horizontal scalability of dynamic analysis through the necessity of acquiring and configuring additional hardware. Consequently, *re-hosting* embedded firmware into a virtual environment has been proven a viable option for dynamic analysis applications [16], [22], [23], [50], [51], [59], [60], [62], [72], [74]. Yet, re-hosting approaches based on emulation of CPU instructions in emulators such as QEMU [11] or Unicorn [57] incur significant emulation overheads.

We introduce *transplantation*, a novel re-hosting technique that executes MCU firmware at the native speed of a more powerful host CPU. Transplantation leverages the key insight that embedded ISAs such as Arm Thumb2 used in Arm Cortex-M based microcontrollers overlap with desktop-grade ISAs as employed in Arm Cortex-A processors. Based on this insight, *transplantation (i)* employs lightweight static binary rewriting to transform bare-metal embedded firmware into a Linux user space program, and *(ii)* natively executes the rewritten firmware alongside a minimal transplantation runtime as a user space process on server-grade hardware. Transplantation only necessitates small adjustments to the target binary to account for semantic differences in target and host ISA, replicate the target's address space appropriately, and intercept peripheral accesses. Our prototype implementation SURGEON accounts for these requirements, and leverages High-level Modeling (HLM) as introduced by HALucinator [16] for peripheral emulation.

Through transplantation, SURGEON provides *(i)* flexibility, ease of use, and scalability through execution as a user space process, *(ii)* high execution speeds due to the native execution of instructions on high-performance CPUs with clock speeds in the range of multiple gigahertz (GHz) instead of at maximum a few hundred megahertz (MHz) as is the case on the original MCU, and *(iii)* precise peripheral emulation through analyst-provided peripheral models. We make SURGEON available as open-source software at https://github.com/HexHive/SURGEON.

Workshop on Binary Analysis Research (BAR) 2024
1 March 2024, San Diego, CA, USA
ISBN 979-8-9894372-0-7
https://dx.doi.org/10.14722/bar.2024.23011
www.ndss-symposium.org

II. Background

In order to motivate the need for transplantation-based embedded firmware analysis, we provide background information on dynamic binary analysis. By summarizing the topic of re-hosting, we bridge the gap between dynamic binary analysis in a generic sense and software analysis for embedded firmware.

A. Dynamic Binary Analysis

Static analysis of software on both source code or binary artifacts lacks runtime information about control flow and internal state from concrete executions. Consequently, static analysis commonly over-approximates potential control flow. Due to this over-approximation and the lack of information on runtime state build-up such as the layout of variables on the heap, static analysis frequently incurs high false positive rates [29], [39], [43].

Dynamic analysis, in contrast, gathers insights about the Program Under Test (PUT)'s behavior during execution. Instrumentation for dynamic analysis can be introduced at source level (e.g., ASAN [63] or AFL++ [24]), binary level (e.g., RetroWrite [20], QEMU [11] or Valgrind [52]), or through hardware tracing features (e.g., Intel Processor Trace [37] or Arm CoreSight [8]). Dynamic binary analysis potentially only covers the actually executed subset of all the code present in the PUT. However, dynamic analysis techniques provide an analyst with insights into the runtime state of an *actual* execution of the target. As an example, fuzz testing [28], [67] has proven to be an effective means of discovering numerous bugs in software ranging from user space programs over Operating System (OS) kernels and hypervisors to different types of firmware [12], [14], [24], [47], [59], [61], [70].

While an analyst can leverage kernel-provided APIs such as `ptrace` and `perf` to gain insights into a user space process' behavior, such interfaces are not available when analyzing bare-metal firmware. Furthermore, instead of relying on the kernel for I/O operations through system calls, MCU firmware directly accesses hardware peripherals to communicate with the outside world. For these reasons, dynamic analysis for embedded firmware commonly leverages *re-hosting*, which we elaborate on in the following.

B. Re-hosting

Re-hosting is the process of building a virtual environment for a given firmware image that sufficiently models the firmware's hardware dependencies [22]. The goal of this process is to design this virtual environment in a way that the behavior of re-hosted firmware in this environment is representative of executions on the real hardware.

In a re-hosted environment, three main components need to be modeled with sufficient precision to achieve the desired representativeness: *(i)* the ISA, *(ii)* hardware peripherals, and *(iii)* additional hardware-specific behavior. This requirement stems from the inherent differences between a system based on an embedded MCU and the analyst's host system.

First, MCUs typically implement simple ISAs such as Arm's M-profile [66], PowerPC [55], or RISC-V [21] to achieve small chip area and low power consumption. Analysts' host systems however may be built around commodity desktop CPUs, commonly implementing the x86(-64) ISA. For this reason, existing re-hosting approaches commonly leverage emulators such as QEMU [11] or Unicorn [57] for ISA translation [16], [23], [50], [59], [60], [74]. This approach introduces *emulation overheads* through ISA translation and software-based address translation between the target and host address spaces. Recently, SafireFuzz [62] exploited compatibilities between Arm Cortex-M and Cortex-A's ISAs to fuzz embedded firmware with high performance. However, SafireFuzz ignores any remaining incompatibilities between the ISAs, lacks modeling of additional MCU-specific behavior, and thus *threatens correctness* of instruction execution and system modeling. Transplantation as shown in Section III ensures semantic correctness of instructions and proper modeling of the target system in a re-hosted environment.

Second, a re-hosted firmware's peripherals must be modeled to account for the firmware's I/O operations. Re-hosting systems either employ Memory-mapped Input/Output (MMIO)-based peripheral modeling or High-level Modeling (HLM) for this purpose. In the first case, the re-hosting system aims to model real peripherals' behavior and handles memory accesses to the MMIO region accordingly. On the one hand, hardware-in-the-loop systems forward memory accesses to the MMIO region via debug probes to real hardware [41], [50], [72], which *hinders scalability* due to the reliance on actual hardware. Systems emulating peripherals in software through heuristic or symbolic-execution-based approximations [23], [59], [60], [74] on the other hand *incur imprecisions* in the peripheral modeling process. HLM, as introduced by HALucinator [16] and employed by other systems [62], in contrast intercepts control flow in the firmware at a pre-defined higher level such as the Hardware Abstraction Layer (HAL) API. Instead of accurately mimicking complex real hardware's behavior, only the HAL API's behavior with known semantics needs to be replicated. As code calling into HAL libraries is oblivious to the logic executing beneath the HAL, replacing HAL functions with developer-provided emulation code does not affect the semantics of the firmware's application logic.

Third, MCU-specific hardware behavior such as configuring, dispatching and handling interrupts needs to be replicated in a re-hosting environment as well. Such functionality in certain cases can already be provided by an underlying system emulator such as QEMU [11] but in other cases needs to be accounted for by the developer of a re-hosting platform if the underlying emulator does not support such functionality out of the box as is the case for example with Unicorn [57].

III. Transplantation

Transplantation is a versatile, performant, and easy-to-use alternative to existing re-hosting approaches. Our key insight is that significant overlaps between certain embedded MCU ISAs and the ISAs supported by server-grade CPUs enable straightforward cross-ISA translation. Exemplarily, the ISA implementation of Arm's Cortex-M lineup of microcontrollers is highly compatible with the ISA implementation employed in the Cortex-A lineup of CPUs as used in mobile, desktop, and server-grade processors [5], [6], [54]. This observation allows natively executing the majority of code of a Cortex-M based MCU firmware in a Linux user space process on a Cortex-A CPU, removing the requirements of ISA emulators and the complexity added by such a layer of indirection. In the following, we describe the challenges that re-hosting faces and showcase how transplantation addresses these challenges to provide flexible and performant re-hosting of embedded firmware. We describe SURGEON, our prototype of a transplantation system, and its potential use cases in Sections IV and V. Throughout the rest of the paper, we use the term *target system* to refer to the MCU-based system the binary was originally intended to be executed on, and *host system* for the system the re-hosted environment is running on.

A. Challenges

Challenge 1 – Representation of the Physical Address Space: MCU firmware statically links all code and global data. MCUs then map this firmware, its heap and stack as well as MMIO-mapped peripherals and control registers into a single flat physical address space. Those different memory regions for ROM, RAM or MMIO are placed at fixed, architecture-determined locations in this physical address space. Re-hosting solutions must account for the corresponding address space structure to accurately model the firmware's expected execution environment.

Hardware-in-the-loop, pure emulation, and HLM approaches all introduce a complex and expensive emulation layer. This layer imposes a software-based address translation for every access to the address space, and dispatches to different emulator logic based on which region in the address space is accessed. For example, accessing the MMIO region requires peripheral-specific handling of loads and stores, whereas accessing RAM requires only Memory Protection Unit (MPU) access permission checks. In summary, all accesses must respect the original MCU's physical address space layout.

Challenge 2 – MCU Architectural Features: MCUs differ architecturally compared to a CPU's user space mode. Such differences include, e.g., execution modes, interrupt controllers (Arm's Nested Vectored Interrupt Controller (NVIC)), banked registers, or hardware-supported exception entry and return routines not present in a Linux process. Especially this last example presents a significant difference between Arm Cortex-M MCUs and an unprivileged Cortex-A user space context. Loading a value into the program counter register always triggers a branch to that location in user space, while loading a magic value (`0xFFFFFFF1` to `0xFFFFFFFD`) instead triggers a hardware-implemented exception return routine on Arm Cortex-M. Re-hosting must account for such features. Previous hardware-in-the-loop [50], [72], HLM [16] or purely emulation-based [23], [59] re-hosting solutions commonly emulate the corresponding logic, managing complex CPU state solely in software. Other approaches executing MCU firmware in user space ignore this challenge altogether [46], [62].

Challenge 3 – Peripheral Handling: MCU firmware interacts with the outside world using peripherals. A re-hosting environment must handle peripheral accesses with varying degrees of accuracy, depending on the analyses to be conducted and how the results of those analyses map to the real hardware. Hardware-in-the-loop approaches are commonly the most accurate due to their usage of ”real” peripherals. Pure emulation provides high accuracy using the peripheral's MMIO interface. However, this approach suffers from significant engineering efforts to correctly implement the vast diversity of available peripherals [22]. HLM leverages the HAL layer of peripherals to reduce the engineering effort for peripheral emulation. However, as a consequence, this approach does not execute low-level peripheral specific code beneath the HAL layer. The HAL interface provides information about the semantics of data being passed in and out of the firmware from and to peripherals, allowing highly accurate modeling of those interfaces. Peripheral modeling based on symbolic modeling or automated peripheral behavior inference exposes a higher degree of automation but fails to guarantee the same level of accuracy as HLM.

Challenge 4 – Correct Execution of Instructions: The execution of instructions must be semantically equivalent to the execution on the original MCU. Semantic equivalence comprises both the instructions' intended action (e.g., arithmetic operations) and potential side-effects such as setting CPU flags. While this generally does not pose an abundant problem, emulators and hypervisors may introduce inaccuracy when modeling a physical CPU [4], [26], [48], impacting semantic correctness of the re-hosting environment.

Challenge 5 – Performance and Scalability: Many dynamic analysis techniques benefit from high execution speeds and horizontal scalability. For example, fuzzing benefits from fast and massively parallelized execution due to more iterations being executed in less time. While not all dynamic analysis use cases strictly require these properties, a re-hosting system aiming to be generic and applicable to a broad range of use cases needs to take performance and scalability into account.

Hardware-in-the-loop approaches emulate the target's ISA and forward peripheral accesses to real hardware via debug probes. They require dedicated MCU hardware for each running firmware instance. Thus, these approaches lack scalability and suffer from performance degradation due to the forwarding of accesses to the physical device. In contrast, pure emulation and High-level Modeling (HLM) approaches do not require dedicated hardware and scale horizontally. Unfortunately, prior systems pay a hefty emulation tax for emulating a foreign ISA.

Challenge 6 – Introspection: Dynamic analysis requires introspection into the Program Under Test (PUT) at runtime. Hardware-in-the-loop, pure emulation, and HLM usually require the presence of a hardware debugging interface or modification of the emulation layer, i.e., adding a debugger stub to enable attaching a debugger. Therefore, these approaches result in additional engineering effort, a restricted feature set in comparison to common interfaces for user space software, communication overhead, or a combination of the above.

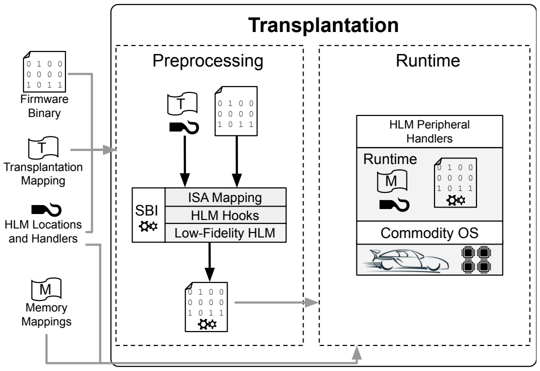


Fig. 1. An overview of transplantation. Key contributions are shown in gray.

B. Transplantation

Firmware *transplantation* addresses the aforementioned challenges. Transplantation provides address space layout replication in a user space process's virtual address space, execution speed and correct instruction semantics through native execution, and introspection capabilities through standard tooling for Linux user space processes. Any behavioral differences that stem from hardware-implemented behavior on an embedded MCU as well as peripherals are modeled in a minimal transplantation runtime.

More specifically, transplantation leverages the insight that many ISAs implemented in low-performance MCUs are also supported either fully or to a great extent in high-performance desktop and server CPUs. Examples for ISAs providing such an overlap are Arm's Thumb2 instruction set employed in the Cortex-M and Cortex-A architectures, PowerPC, or RISC-V [9], [21], [34], [54], [65], [66], [68]. Transplantation as a concept is architecture-agnostic and only requires this syntactic and semantic compatibility of target and host ISAs to natively execute MCU firmware binaries in a Linux user space process on a compatible ISA. This approach differs from both *binary ISA translation* and *direct execution* of firmware binaries and provides an analyst with multiple advantages. *Binary ISA translation* describes the idea of translating from one execution environment and ISA to a fundamentally different combination of ISA and environment, such as re-hosting an Arm Cortex-M firmware into an emulator on an x86-based host machine. *Direct execution* on the other hand describes the execution and analysis of targeted software on the original device. We position transplantation as a middle ground between these two approaches. Transplantation leverages the native ISA of the target firmware and replicates the rest of the environment in a minimal user-space runtime. Consequently, we obtain the correctness of instruction execution provided by direct execution, and the flexibility of a software-modeled execution environment typically provided by emulators. Further, transplantation overcomes the ISA translation overheads of emulators such as QEMU and the performance limitations of MCUs with low clock frequencies (few MHz) by natively executing firmware code in a user space process on a fully-featured high-performance CPU (multiple GHz).

Transplantation follows a two-step approach: First, after ensuring syntactic compatibility of the host and target ISA, we transform the target binary into a user-space-compatible program via Static Binary Instrumentation (SBI). In this step, we *(i)* address any potential semantic differences between the two involved ISAs, *(ii)* insert branches into the runtime to HLM peripheral handlers for complex HAL functions, *(iii)* replace simple HAL functions inline with corresponding emulation code (low-fidelity HAL), *(iv)* and optionally insert additional analyst-defined instrumentation. In the second step, our runtime ensures correct execution of the target firmware in a user space process. The runtime replicates the target environment by loading the firmware binary into the virtual address space at the defined locations, thereby maintaining the expected address space layout. Moreover, the runtime emulates hardware-specific behavior that cannot be addressed during static binary rewriting in software and redirects any peripheral accesses to the corresponding HLM emulation functions. Figure 1 shows this two-stepped approach. In the following sections, we detail the design of the instrumentation and runtime phases, respectively.

1) Static Binary Instrumentation: We first analyze and modify the provided binary to ensure interoperability with our minimal Linux user space process runtime and syntactic and semantic compatibility of the target's and host's ISAs. Transplantation exploits the overlap between target and host system ISAs. Any remaining differences in instruction semantics between the target and the host system are accounted for by replacing problematic instructions in the binary. These differences typically stem from architecture-specific behavior, such as interrupt and privilege level configuration or coprocessor accesses. Any replacements in the binary are done *(i)* inline, *(ii)* by inserting direct branches to emulation code, or *(iii)* by replacing the instruction with a trapping instruction and emulating the logic in a trap handler in the runtime. To maintain the original address space layout (Challenge 1), replacements must not exceed the replaced code in size. In case of space limitations that prevent inline replacement of instructions with semantically equivalent code snippets, instructions are replaced with branches or traps to emulation code. The aforementioned approaches are thus listed in order of decreasing preference, with increasing performance impact and at the same time increasing general applicability. While inline replacements are possible in many cases, trapping into the runtime is a fail-safe fallback. Any required additional code for branch targets or trap handlers is added in unused parts of the address space (e.g., reserved regions in the physical address space that an MCU firmware would never refer to). Through this approach, the target system's architectural features that are not present on the host system are faithfully replicated in software, addressing Challenge 2. Any non-modified code and data in the target binary are copied verbatim into the output binary. Consequently, the target binary's layout is unmodified, ensuring proper representation of its address space (Challenge 1). Additionally, branches to handlers in the runtime are inserted at the beginning of each function that is to be replaced via High-level Modeling (HLM). In the embedded system context, this allows a transplantation system such as SURGEON to intercept and redirect calls to HAL functions during execution, and provides the necessary prerequisite for successful peripheral emulation via HLM (Challenge 3). We further describe the runtime component of this method of emulating peripherals in Section III-B2.

2) Runtime: Our runtime first loads the target binary into its virtual address space. In this process, the runtime ensures that Challenge 1, the correct replication of the expected address space layout is addressed by mapping code and data into the virtual address space at their expected locations. The runtime's code and data itself are located in unused parts of the virtual address space in order to not interfere with the Program Under Test (PUT)'s memory ranges. Apart from this initial loader, the runtime contains the code for emulating device-specific behavior such as exception entry and return routines provided by hardware in the original MCU, as well as any handlers emulating peripheral accesses via HLM. Thereby addressing Challenges 2 and 3, this emulation code is executed whenever the target firmware reaches code locations that were previously instrumented for replicating MCU-specific behavior or peripheral accesses via HLM. After loading the PUT into the virtual address space, the runtime hands over control to the target program through a branch to its designated entry point. From this point on, the re-hosted firmware executes natively on the host CPU. This execution model provides transplantation with the high performance stated as a goal in Challenge 5, thanks to native code execution on a high-performance host CPU. The re-hosted system's state can be easily replicated by forking the user space process, allowing for simple horizontal scalability across cores in the host system. Furthermore, executing natively in a user-space process provides an analyst with a plethora of mature tooling for introspection and analysis (Challenge 6). For example, Linux's `ptrace` API enables the analyst to debug a re-hosted PUT with standard debuggers such as GDB without the need for a potentially limited GDB server stub implementation in an emulator. Additionally, this approach enables the analyst to leverage unmodified user space tooling such as AFL++ [24]. We employ the latter for fuzzing embedded firmware as further described in Section V-B. Transplantation consequently provides an analyst with full flexibility in terms of applicable user space tooling without needing to modify the tools for a different environment in an emulator or on real hardware.

IV. SURGEON: RE-HOSTING VIA TRANSPLANTATION

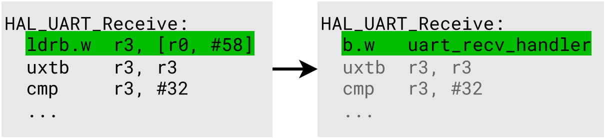
We present SURGEON, a transplantation-based re-hosting system targeting Arm Cortex-M embedded firmware. Our prototype re-hosts such firmware into a Linux user space process natively executing on Arm Cortex-A chips. In the following, we provide details on the implementation of SURGEON's two main components, the Static Binary Instrumentation (SBI) engine as well as our runtime component.

1) Static Binary Instrumentation: We implement SURGEON's SBI engine in Python, leveraging Capstone [1] and Keystone [2] for disassembling and assembling code, respectively. As a first step, we employ Ghidra [27] to identify basic block boundaries for potential instrumentation during static binary rewriting. We feed the set of identified basic blocks as well as a list of explicit instrumentation targets to the SBI engine, which in turn leverages this information to instrument the given firmware binary. Our instrumentation engine supports multiple instrumentation passes with different targets and granularity. In a first pass, our SBI engine addresses semantic incompatibilities between the target and host ISAs. In our prototype, we re-host Arm Cortex-M MCU firmware into an Arm Cortex-A CPU Linux user space process. While the majority of instructions can be executed as-is, a small number of instructions require dedicated handling. We categorize these instructions as follows:

Modifying MCU-specific CPU state: The instructions `msr` and `mrs` copy information from general-purpose registers to model-specific registers and vice versa, respectively. If possible, we rewrite those instructions to only change arguments with semantic equivalence (e.g., model the Cortex-M Application Program Status Register (APSR) with Cortex-A's Current Program Status Register (CPSR)). Otherwise, we emulate their behavior in other instructions (e.g., copies to/from unused floating point registers for modeling banked stack pointer registers). Where complex logic prevents direct replacement, we replace the instruction with a branch to emulation code or, if dictated by space limitations, a `bkpt` instruction to trap-and-emulate in our runtime.

Software interrupts: The `svc` instruction on both Arm Cortex-M and Cortex-A causes a software interrupt to be raised. Whereas the firmware expects the corresponding interrupt handler to be executed, such a software interrupt in a Linux user space process causes a transition to the kernel and execution of a syscall. In order to execute the firmware's interrupt handler and not issue unexpected syscalls during dynamic analysis, we replace `svc` instructions with `bkpt` instructions and pass control on to the interrupt handler from our runtime's trap handler triggered as a result of the `bkpt`.

Coprocessor accesses: On Arm MCUs, up to 16 coprocessors can be defined for adding features to the main processor. Any instruction executing on the main CPU is also streamed to a coprocessor and executed on the coprocessor if necessary [7]. Hardware floating point support implemented on a coprocessor does not require special handling by SURGEON, since the corresponding floating point instructions are transparently executed on the floating-point implementation of the host Cortex-A CPU. In our experiments (see Section V), we did not encounter custom coprocessors. In case a custom coprocessor is employed, the instructions for coprocessor data processing (`cdp/cdp2`), moving data from Arm registers to the coprocessor and vice versa (`mcr/mcr2`, `mcrr/mcrr2`, `mrc/mrc2`, `mrrc/mrrc2`) and loading and storing data from or to memory (`ldc/ldc2`, `stc/stc2`) need to be rewritten. Depending on the coprocessor's functionality, those



```
HAL_UART_Receive:
  ldrb.w   r3, [r0, #58]
  uxtb     r3, r3
  cmp      r3, #32
  ...
```

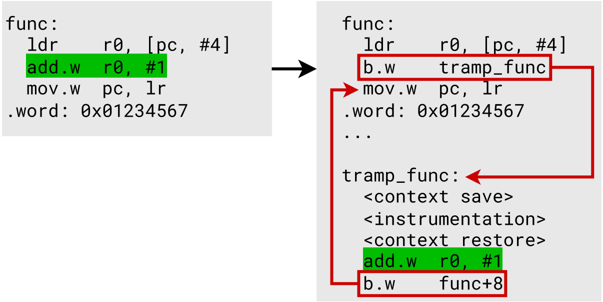
```
HAL_UART_Receive:
  b.w    uart_recv_handler
  uxtb   r3, r3
  cmp    r3, #32
  ...
```

Fig. 2. HAL function entry points are replaced with branches to our handlers. The remainder of the function stays unmodified but is not executed anymore.

instructions are rewritten inline with equivalent functionality, or in case of space limitations are replaced with a call or trap into the runtime for emulation.

After the pass ensuring semantic fidelity, we leverage both generic basic block instrumentation as well as explicitly specified instrumentation points. We instrument the entry points of HAL functions to dispatch to HAL handlers in our runtime. This instrumentation pass simply replaces the function prologue of a HAL function in the firmware with an unconditional branch to the corresponding handler as exemplarily shown in Figure 2. In case a HAL function does not require complex logic to be executed in a handler function, SURGEON patches the function inline (low-fidelity HAL). For example, peripheral initialization functions can typically just return with a value that indicates successful initialization. Additionally, we instrument the previously identified basic blocks in the firmware. For each basic block, we identify instructions that are not program counter (PC)-relative and can thus be relocated without affecting their semantic validity. We replace the first such instruction with an unconditional branch to a trampoline at an unused location in the address space. This trampoline 1) saves firmware context according to the Arm calling conventions, 2) executes instrumentation code, 3) restores context, 4) executes the original instruction(s) replaced by a branch, and 5) finally branches back to the original code location. Figure 3 provides an example of an instrumented control flow. We use this mechanism to reproducibly simulate time in the re-hosted environment. We advance a virtual clock by adding callbacks into the instrumentation code. Virtual timekeeping is required in order to reproducibly trigger interrupts in the firmware at fixed intervals. Through this instrumentation, the reproducibility of time as perceived by the re-hosted firmware only depends on the reproducibility of the control flow and executed basic blocks in the firmware. As we detail in one of our exemplary use cases in Section V-B, an analyst can add additional instrumentation passes with low effort.

2) Runtime: SURGEON's runtime is implemented in C to provide us with full control over the address space layout and performance-critical optimizations. The runtime is linked statically to prevent the dynamic loader from loading libraries at addresses we do not control. All code, data and runtime variables such as stack and heap are located in the vendor-reserved region of the Arm Cortex-M memory map (`0xE0100000–0xFFFFFFFF`) [7]. This approach ensures no collisions with addresses used by re-hosted firmware. After loading the firmware binary at its expected addresses into the virtual address space, the runtime branches to the firmware's entry point and from there on natively executes the PUT.



```
func:
  ldr     r0, [pc, #4]
  add.w   r0, #1
  mov.w   pc, lr
.word: 0x01234567
```

```
func:
  ldr     r0, [pc, #4]
  b.w     tramp_func
  mov.w   pc, lr
.word: 0x01234567
...

tramp_func:
  <context save>
  <instrumentation>
  <context restore>
  add.w   r0, #1
  b.w     func+8
```

Fig. 3. SURGEON adds instrumentation in trampolines.

Besides loading the firmware and handing over control, the runtime contains emulation code for architectural features, trap handlers for the trap-and-emulate approach where required, and handlers for HAL functions. The runtime contains the callback functions that are invoked in the trampolines inserted during static binary instrumentation. This includes, for example, the code incrementing the virtual clock that provides time to the firmware. The trap handlers are implemented via Linux signal handlers. A `svc` instruction replaced with a `bkpt` during rewriting causes Linux to deliver a `SIGTRAP` signal to the process. In the corresponding signal handler, SURGEON's runtime emulates Arm's hardware-implemented exception entry routine in software, branches to the Interrupt Service Routine (ISR) in the firmware responsible for handling the software interrupt, and upon return from that routine emulates an exception return routine.

Finally, the runtime contains the handlers for HAL functions interacting with peripherals in the firmware. Our prototype provides a developer with two implementation options for HAL handlers. First, our runtime incorporates a Python interpreter and a native module exposing the same API as implemented in HALucinator [16] to Python code. Consequently, an analyst can leverage HALucinator's existing HAL handlers for peripheral emulation and quickly prototype new handlers in Python. Second, our prototype also supports calling into native handlers. This approach enables direct branches into the handlers from the HAL functions in the firmware binary when following the original function's prototype thanks to the Arm Architecture Procedure Call Standard (AAPCS) [10]. In this case, the function's arguments have already been set up by the HAL function's caller, and are not affected by the direct branch. When the handler returns, control flow continues directly at the HAL function's call site just as if the original HAL function had returned. In contrast to the Python handlers, this implementation does not require saving and restoring the firmware's context before and after branching into the Python interpreter in addition to the interpreter's own overhead in comparison to native code. During our experiments described in Section V, we observed an on average 5x speedup by using native handlers instead of Python-based implementations. We, therefore, conclude that leveraging HALucinator's existing

HAL handlers in Python provides an excellent base for quick prototyping, but an analyst should implement handlers natively for performance-critical use cases.

V. Applications of Surgeon

Transplantation is a flexible and versatile approach to dynamic analysis. In our prototype implementation, we leverage Surgeon to re-host Arm Cortex-M firmware into a Linux user space process executing on Arm Cortex-A CPU cores. We describe how Surgeon can be leveraged by an analyst for debugging with off-the-shelf user space debugging tools in Section V-A. Surgeon through its native user-space nature greatly simplifies such a use case in comparison to interfacing with emulators or real hardware. Furthermore, we extend the basic binary rewriter with a pass that adds coverage instrumentation for fuzzing, and show that transplantation enables fuzzing with high performance in comparison to emulation-based fuzzers. We detail this use case in Section V-B.

A. Debugging Embedded Firmware

Surgeon simplifies debugging embedded firmware by enabling an analyst to use common user space debuggers for this task. More specifically, an analyst invokes Surgeon's runtime with the re-hosted firmware through the GDB debugger, and steps through the re-hosted firmware like any other user space program. Debugging transplanted firmware benefits from multiple differences to on-device or emulator-based debugging. First, transplantation improves *observability* of memory corruption bugs. The user space nature of software transplanted with Surgeon causes accesses to unmapped memory or with incorrect permissions to result in a segmentation fault. This behavior provides a clear signal for the presence of a bug. On real hardware, the absence of fine-grained memory protection via an MMU may mask illegal memory accesses and cause any effects of such an unintended access, if present at all, to only surface much later in the execution of the firmware [51]. Even the presence of a Memory Protection Unit (MPU) does not necessarily improve upon the observability of memory corruption bugs, since usage of MPUs in COTS firmware is low and their implementation often incorrect [75].

Second, transplantation with Surgeon allows an analyst to *reproducibly* execute the same code multiple times. For example, the analyst cannot fully control the timing and order of interrupts on real hardware. Consequently, the control flow of each execution of the firmware may differ across executions. Surgeon with its reproducible virtual clock based on basic-block instrumentation ensures the same timing and order of interrupts for each execution of the transplanted firmware and hence prevents *heisenbugs* which cannot be reproducibly observed across executions.

Finally, debugging via Linux' `ptrace` API that GDB uses speeds up finding potential issues in comparison to emulator-based or on-device debugging. More specifically, our approach can make use of the host CPU's hardware-supported breakpoints and memory watchpoints while executing code at native speed. Leveraging an emulator on the other hand requires the emulator to check every instruction for memory accesses to watchpoints defined by the analyst in software, greatly slowing down execution speeds and therefore hindering the analyst from efficient and fast debugging. Additionally, emulators may provide limited GDB server stub implementations, or none at all. For example, the Unicorn ISA emulator [57] used by hal-fuzz and Fuzzware by default does not come with a GDB server implementation at all. Therefore, any debugging features need to be implemented on top of the emulator by an analyst. Similarly, on-device debugging requires the presence of debug interfaces such as JTAG which are commonly omitted in COTS devices. Surgeon meanwhile enables usage of the full spectrum of GDB's features through user space debugging.

B. EmbedFuzz

As a proof of concept for the flexibility of the transplantation approach, we implement EmbedFuzz, an extension of Surgeon to embedded firmware fuzzing. Previous work such as P2IM [23], the fuzzing-oriented version of HALucinator hal-fuzz [16], Fuzzware [59] or Hoedur [60] are based on emulators and require extensive modifications to emulators, existing fuzzing engines, or both. Surgeon on the other hand thanks to its native user-space nature can be combined with off-the-shelf fuzzers such as AFL++ [24], [33]. In addition, native execution on the host CPU removes any overhead stemming from cross-ISA instruction translation and other side effects of emulation such as software-based address translation.

As mentioned in Section IV-1, Surgeon enables an analyst to easily specify additional instrumentation passes. We leverage this fact to add coverage instrumentation to each basic block in the target firmware for coverage-guided fuzzing. As an optimization, we also add a command-line flag to our runtime allowing the analyst to choose whether to only execute the re-hosted firmware once or in a fork server mode compatible with AFL++. With those minimal additions to Surgeon, EmbedFuzz enables high-speed fuzzing without emulation overheads with an unmodified version of AFL++.

To demonstrate EmbedFuzz's capabilities, we evaluate our implementation on 10 real-world MCU firmware binaries introduced in P2IM [23]. We run fuzzing campaigns for EmbedFuzz and the state-of-the-art firmware fuzzers Fuzzware [59] and P2IM. Hal-fuzz [16] and SafireFuzz [62] both lack crucial architectural features which are present in this firmware data set such as support for software interrupts via the `svc` instruction. Due to these incompatibilities, we could not run fuzzing campaigns with these systems. In order to showcase the potential performance impact of ISA emulators, we also run fuzzing campaigns where the re-hosted firmware is emulated via `qemu-user`. We conduct ten 24-hour fuzzing campaigns per fuzzer and report minimum, average and maximum numbers for our evaluation results. We execute EmbedFuzz natively on a SolidRun Honeycomb board built around an NXP Layerscape LX2160A processor featuring 16 Arm Cortex-A72 cores [54] and 32GB of RAM. The dependencies of the other evaluated fuzzers require running

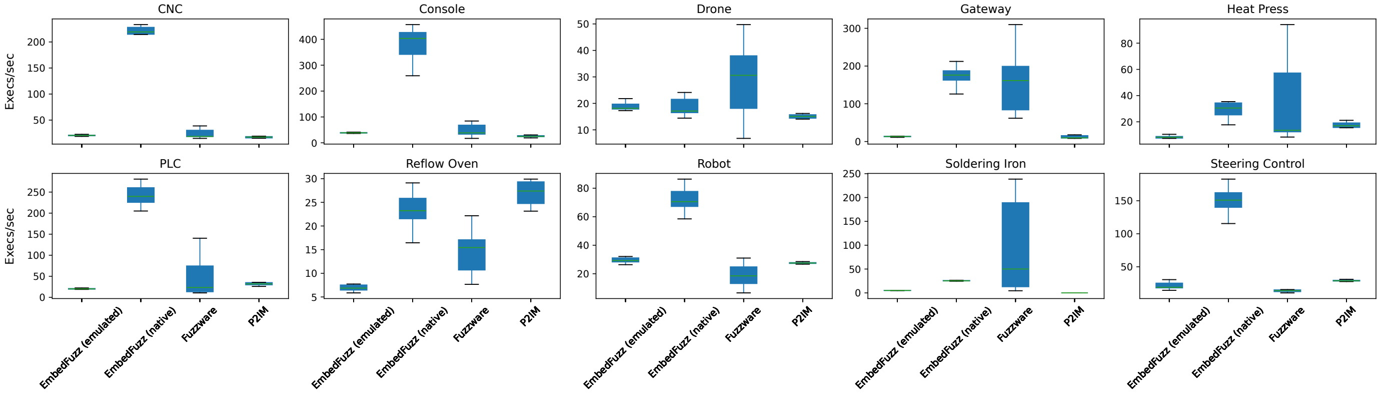


Fig. 4. Box plot of fuzzing executions per second (i.e., throughput) on real-world firmware binaries across ten 24 hour runs.

on the Intel x86-64 ISA. Hence, we conduct the fuzzing campaigns for the emulated version of EmbedFuzz, Fuzzware and P2IM on servers based on Intel's Xeon Gold 5128 [36] and 64GB of RAM. Both systems run Ubuntu 22.04.

Figure 4 shows the fuzzing throughput for the aforementioned four types of fuzzing campaigns. Fuzzing throughput is crucial for fuzzing campaigns, and transplantation helps to improve execution performance when re-hosting targets that would otherwise require running in an emulator. EmbedFuzz exhibits at least on-par throughput and up to 8x speedup in comparison to Fuzzware and P2IM. The comparison of native and emulated execution of our runtime and firmware shows on average seven times higher fuzzing throughput. Notable exceptions to these improvements are the Reflow Oven and Soldering Iron firmware. The Reflow Oven firmware heavily utilizes time measurements during which the firmware loops until an internal counter reaches a certain value. P2IM's QEMU-based virtual clock implementation handles such cases better than those of Fuzzware and EmbedFuzz, resulting in higher throughput. The Soldering Iron firmware on the other hand is based on FreeRTOS [3] which leverages software interrupts via `svc` for switching between tasks at high frequency. Our trap-and-emulate approach described in Section IV-1 causes overheads due to context switches in and out of the host kernel during trap dispatching, reducing the potential performance advantage of EmbedFuzz in this specific case.

We provide the coverage over the 24-hour fuzzing campaigns for the firmware dataset we evaluate in Figure 5. EmbedFuzz achieves higher coverage than P2IM in all ten cases, and is on par with or improves upon Fuzzware in 5 out of 10 cases. For Fuzzware, we exclude basic blocks below the HAL when calculating the coverage for a fair comparison with EmbedFuzz. Due to the HLM approach of handling peripherals, any code beneath the HAL functions redirected to the handlers in our runtime is not reachable by the fuzzer. For all the firmware, EmbedFuzz quickly achieves its maximum coverage before stagnating. Manual investigation revealed conditional guards that are difficult to bypass with brute-force guessing. Overcoming hurdles such as magic values and checksums is out of scope for this paper and can be added on top of EmbedFuzz by an analyst thanks to the flexibility in instrumentation passes and peripheral handler implementations described in Section IV. Beyond the performance improvements, transplantation also improves upon the observability of firmware crashes. As explained in the previous section, the user space nature of transplanted firmware causes memory accesses with missing or incorrect permissions to result in a segmentation fault, crashes the process and is used by the fuzzer as a signal for a potential bug. During our fuzzing campaigns with EmbedFuzz, SURGEON simplified triaging and debugging crashes based on the advantages of transplantation for debugging described in Section V-A.

VI. DISCUSSION

Even though transplantation as a concept and its implementation in SURGEON as highlighted in the previous sections provide analysts with a performant and flexible way of dynamically analyzing MCU firmware binaries, we acknowledge limitations to its current implementation. In the following, we discuss those limitations and provide potential paths forward for alternative or additional approaches.

High-level Modeling: Transplantation in general does not dictate how the target software communicates with the outside world. In our implementation of SURGEON, we focus on re-hosting Arm Cortex-M firmware which communicates with other devices via hardware peripherals. We model those peripherals following HALucinator's [16] idea of HLM and inherit limitations of this approach. Notably, we require re-hosted firmware to leverage HAL libraries or similar high-level APIs in order for SURGEON to intercept and emulate peripheral accesses at this interface. Furthermore, SURGEON does not require source code for the full firmware for successful re-hosting. However, we require source code availability for the HAL libraries in order to correctly identify the locations of HAL functions in the binary either through provided symbols or HALucinator's LibMatch approach. The latter correlates HAL functions compiled from source code with code in the firmware binary to locate HAL functions' entry points.

We argue that the general availability of Hardware Abstraction Layer (HAL) library source code through MCU

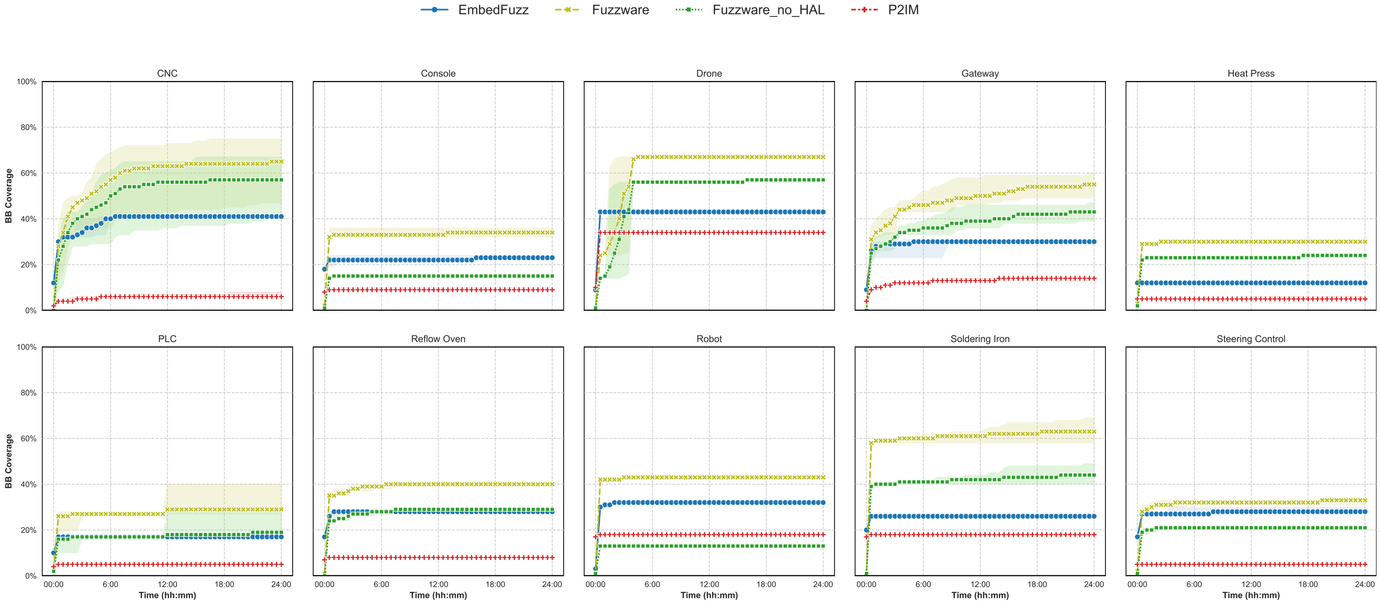


Fig. 5. Code coverage over time for real-world firmware binaries across ten 24 hour trials. The median percentage of uniquely discovered basic blocks, as well as the minima and maxima for each trial are illustrated.

vendor SDKs results in widespread use in real-world deployments. Therefore, High-level Modeling (HLM) enables reuse of peripheral models across firmware as long as they share a HAL implementation that is already supported in SURGEON. Implementation effort of peripheral handlers is hence reduced to a one-time undertaking with great reusability.

The general principle of transplantation permits alternative implementations. For example, an analyst could leave the MMIO region unmapped in the process's virtual address space so that MMIO accesses cause segmentation faults. The peripherals backing the accessed MMIO registers could then be emulated by dispatching from a signal handler in the process to the corresponding emulation logic based on the address that was read from or written to. However, the context switches for signal dispatching could prevent such an approach from applying to use cases with high-performance requirements such as fuzzing.

Generality of transplantation: SURGEON's transplantation approach benefits from performance improvements through native execution instead of emulation. As described in Section III, this requires a host CPU compatible with the target CPU's ISA. In our implementation, we re-host Arm Cortex-M firmware into a user space process running on Arm Cortex-A CPUs. Our implementation is also compatible with firmware targeting Arm Cortex-R systems as long as it does not leverage a MMU. Our implementation replicates the firmware's physical address space in the process's virtual address space which is incompatible with the virtual addressing enabled by an MMU in the target system, which would require an additional software-based address translation layer.

SURGEON's transplantation approach can certainly be extended to other architectures as well. For example, the PowerPC ISA is featured both in MCUs (e.g., NXP's line of MPC5xxx MCUs [55]) as well as high-performance server CPUs (e.g., IBM's Power10 CPUs [35]). The same holds for the RISC-V ISA, which is employed by multiple manufacturers and chip designers such as SiFive, Espressif or Alibaba's T-Head both for embedded MCU and high-performance server designs [21], [65], [68]. User space software purely running in a virtual address space can also be instrumented and run on a higher-performance CPU, for example by leveraging the compatibility of 32bit and 64bit code on CPUs based on Intel's x86 ISA. As an example, memTrace [56] hides memory tracing instrumentation for 32bit x86 binaries in the upper part of a 64bit process's address space, since only the lowest 4GB are addressable by 32-bit executables. However, in other cases such as firmware targeting the XTensa or AVR embedded ISAs, transplantation is not applicable due to the lack of high-performance CPUs supporting the corresponding ISA.

Transplantation relies on the host CPU supporting a superset of the target CPU's features. Our implementation of SURGEON currently does not support Arm Cortex-M specific features such as Memory Protection Units (MPUs). SURGEON could be extended to support such features by emulating them in software at a performance cost.

SURGEON's static binary rewriting approach only targets code known at instrumentation time. If firmware dynamically loads additional code modules through peripherals or employs self-modifying code, our rewriter cannot target such code and is therefore unable to instrument it.

VII. RELATED WORK

Similar to our prototype implementation of SURGEON, SafireFuzz [62] proposes executing Arm Cortex-M firmware natively on Arm Cortex-A CPUs. However, SafireFuzz's dynamic binary rewriting approach at runtime is purely targeting fuzzing use cases, whereas SURGEON's combination of static binary rewriting and a minimal runtime generically transplants firmware into a Linux user-space process for any dynamic analysis use case such as debugging or dynamic data flow analysis. Furthermore, SafireFuzz executes any instruction as-is without ensuring semantic equivalence on the host CPU. Such an implementation threatens the semantic correctness of executed code, and we deem this system consequently not suitable for the analysis of firmware that contains any of the instructions described in Section III.

Para-rehosting [46] relies on source code availability and compiles embedded firmware source code together with manually implemented peripheral handlers into a user space binary. However, firmware source code for COTS IoT devices is typically not made publicly available by the manufacturer. SURGEON in contrast works on firmware binaries and hence inherently supports more targets.

Other dynamic analysis approaches for embedded firmware are commonly built on top of ISA emulators such as QEMU [11] and Unicorn [57] and mainly differ in their approach to handling peripheral accesses. Hardware-in-the-loop solutions [15], [17], [30], [41], [42], [45], [50], [69], [72] emulate the target firmware's ISA and forward peripheral access via debug interfaces to the actual MCU. This approach suffers from communication overheads with the MCU through debug probes and is limited in scalability due to acquisition and configuration requirements of additional hardware. SURGEON emulates peripherals in software and scales by instantiating additional lightweight Linux user space processes.

Symbolically [13], [32], [59], [60], [64], [74] or heuristically [23], [31], [49] modeling peripheral behavior based on MMIO accesses enable fully automated firmware re-hosting. However, both approaches risk over-approximating peripheral capabilities as previously discussed by Fasano et al. [22]. Further, these solutions require costly upfront analysis of peripheral behavior before starting the intended dynamic analysis task. SURGEON in its current implementation requires a one-time developer effort for implementing peripheral handlers at the HAL level but provides higher accuracy thanks to domain-expert knowledge during implementation.

Finally, other re-hosting approaches model peripherals in software at the kernel level, communicating with user space software through a generic POSIX API [14], [18], [38], [44], [73]. These systems target Linux-based firmware and leverage both kernel and user space and are hence not compatible with SURGEON's current focus on bare metal MCU firmware. Such Linux-based firmware is not suitable for re-hosting via transplantation because transplantation leverages the host's MMU for efficiently replicating the target's physical address space in a host process's virtual address space. Linux-based firmware leveraging an MMU itself requires another layer of indirection for virtual address translation in software, which is outside of the scope of this paper.

VIII. CONCLUSION

SURGEON is the first general-purpose re-hosting system for targets such as MCU firmware running the targets at native speed. We introduce *transplantation* as a novel concept for re-hosting, providing a fast alternative to emulation-based approaches while at the same time simplifying the software stack for arbitrary dynamic analysis use cases. Transplantation leverages compatibilities of low-power target and high-performance host ISAs, such as Arm Cortex-M and Cortex-A, respectively, and addresses any remaining incompatibilities in software. Our SURGEON prototype employs and expands upon High-level Modeling (HLM) for peripheral modeling. We generically re-host MCU firmware, both for single-execution use cases such as debugging as well as high-performance use cases requiring rapid successive executions of the firmware such as fuzzing. EmbedFuzz as a fuzzing-oriented extension of SURGEON outperforms state-of-the-art fuzzers in terms of performance by a factor of up to eight times the fuzzing throughput. In our design and implementation, we stress the importance of semantically correct execution of target instructions which SURGEON achieves through transplantation. Consequently, SURGEON is a fast, flexible, and precise re-hosting system for arbitrary dynamic analysis use cases.

IX. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback on the paper. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), DARPA HR001119S0089-AMP-FP-034, and NSF CNS-1942793.

REFERENCES

[1] "Capstone: The Ultimate Disassembly Framework," accessed: January 2024. [Online]. Available: https://www.capstone-engine.org/

[2] "Keystone: The Ultimate Assembler." [Online]. Available: https://www.keystone-engine.org/

[3] Amazon Web Services, Inc, "FreeRTOS," accessed: July 2022. [Online]. Available: https://www.freertos.org/

[4] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual CPU Validation," in *Proceedings of the 25th Symposium on Operating Systems Principles*. Monterey California: ACM, Oct. 2015, pp. 311–327.

[5] Ampere Computing, "Ampere® Altra®," 2022. [Online]. Available: https://amperecomputing.com/processors/ampere-altra/

[6] Apple Inc., "Mac – Apple," 2022, accessed: July 2022. [Online]. Available: https://www.apple.com/mac/

[7] Arm Limited, "Armv7-M Architecture Reference Manual," Tech. Rep., 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0403/ee/?lang=en

[8] ——, "Arm CoreSight Architecture Specification," Tech. Rep., 2022. [Online]. Available: https://developer.arm.com/documentation/ihi0029/f/?lang=en

[9] ——, "Cortex-M Series Processors," 2022, accessed: January 2022. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m

[10] ——, "Procedure Call Standard for the Arm® Architecture," Tech. Rep., Oct. 2022. [Online]. Available: https://github.com/ARM-software/abi-aa

[11] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[12] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bulekov

[13] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation," in *Annual Computer Security Applications Conference*. Austin USA: ACM, Dec. 2020, pp. 746–759.

[14] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016.

[15] 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 *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_01A-1_Chen_paper.pdf

[16] 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 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[17] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-Wide Security Testing of Real-World Embedded Systems Software," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 309–326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[18] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, X. Chen, X. Wang, and X. Huang, Eds. ACM, 2016, pp. 437–448.

[19] Digi-Key, "MCUs in Industrial Automation," May 2013. [Online]. Available: https://www.digikey.com/en/articles/mcus-in-industrial-automation

[20] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1497–1511.

[21] Espressif Systems, "ESP SoCs," Dec. 2023. [Online]. Available: https://www.espressif.com/en/products/socs

[22] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "SoK: Enabling Security Analyses of Embedded Systems via Rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. Virtual Event Hong Kong: ACM, May 2021, pp. 687–701.

[23] 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 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1237–1254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[25] N. Five, "Withings Body Cardio Teardown," Jan. 2017, accessed: January 2022. [Online]. Available: https://www.ifixit.com/Teardown/Withings+Body+Cardio+Teardown/74987

[26] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui, "HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 366–378.

[27] Ghidra Contributors, "Ghidra Software Reverse Engineering Framework," National Security Agency, Nov. 2022. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[28] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020.

[29] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Transactions on Software Engineering*, pp. 1–37, 2023.

[30] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the Analysis of Embedded Firmware Through Automated Re-Hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson

[31] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 789–806. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison

[32] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. B. Butler, "FirmUSB: Vetting USB device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2245–2262.

[33] M. Heuse, H. Eißfeldt, A. Fioraldi, D. Maier, and J. Aydinbas, "The AFL++ fuzzing framework," 2022, accessed: July 2022. [Online]. Available: https://aflplus.plus/

[34] IBM, "IBM power S1022 servers," 2022, accessed: October 2022. [Online]. Available: https://www.ibm.com/products/power-s1022

[35] ——, "IBM power10: Engineered for agility," 2022, accessed: January 2022. [Online]. Available: https://www.ibm.com/it-infrastructure/power/power10

[36] Intel Corporation, "Intel® Xeon® Gold 5218 Processor Product Specification," 2017, accessed: January 2022. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html

[37] ——, *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, Jun. 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html

[38] M. Jiang, L. Ma, Y. Zhou, Q. Liu, C. Zhang, Z. Wang, X. Luo, L. Wu, and K. Ren, "ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 734–748.

[39] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 672–681.

[40] A. Joshi, "Philips hue: Setup and teardown," 2013, accessed: January 2022. [Online]. Available: https://allthingscc.wordpress.com/2013/01/21/philips-hue-setup-and-teardown/

[41] M. Kammerstetter, D. Burian, and W. Kastner, "Embedded security testing with peripheral device caching and runtime program state approximation," in *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[42] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. Kyoto Japan: ACM, Jun. 2014, pp. 329–340.

[43] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 698–709.

[44] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis," in

ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020. ACM, 2020, pp. 733–745.

[45] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*, A. Francillon and T. Ptacek, Eds. USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher

[46] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021.

[47] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "ViDeZZo: Dependency-aware virtual device fuzzing," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 3228–3245.

[48] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, pp. 337–348.

[49] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of DMA input channels for dynamic firmware analysis," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1938–1954.

[50] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^{2}: A multi-target orchestration platform," in *Proceedings 2018 Workshop on Binary Analysis Research*, vol. 18. San Diego, CA: Internet Society, 2018.

[51] 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 *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018.

[52] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '07*. San Diego, California, USA: ACM Press, 2007, p. 89.

[53] NOVO Engineering, "Embedded Design for an Automated Insuline Delivery System." [Online]. Available: https://novoengineering.com/portfolio/insulin-pump-controller-embedded/

[54] NXP Semiconductors, "NXP Layerscape LX2160A, LX2120A, LX2080A Data Sheet," 2020, accessed: January 2022. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-lx2160a-lx2120a-lx2080a-processors:LX2160A

[55] ——, "MPC5xxx microcontrollers," 2022, accessed: January 2022. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/power-architecture/mpc5xxx-microcontrollers:POWER_ARCH_5XXX

[56] M. Payer, E. Kravina, and T. R. Gross, "Lightweight Memory Tracing," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 115–126. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/payer

[57] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat USA*, vol. 476, 2015. [Online]. Available: https://www.unicorn-engine.org

[58] I. Roseman, "Omnipod DASH insulin pump teardown," Oct. 2022. [Online]. Available: https://idoroseman.com/omnipod-dash-insulin-pump-teardown/

[59] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[60] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "Hoedur: Embedded firmware fuzzing using multi-stream inputs," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/scharnowski

[61] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2597–2614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[62] L. Seidel, D. Maier, and M. Muench, "Forming Faster Firmware Fuzzers," in *USENIX Security*, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/seidel

[63] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[64] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware

[65] SiFive, Inc., "RISC-V Core IP Portfolio," Dec. 2023. [Online]. Available: https://www.sifive.com/risc-v-core-ip

[66] STMicroelectronics, "STM32 Arm Cortex MCUs - 32-bit Microcontrollers," 2022. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[67] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007.

[68] T-Head Semiconductor, "XuanTie Processor IP," Jan. 2024. [Online]. Available: https://www.t-head.cn/product/overview?spm=a2ouz.12986968.0.0.61d3138464xuSV

[69] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 291–307. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

[70] D. Vyukov and A. Konovalov, "Syzkaller: An unsupervised coverage-guided kernel fuzzer," Google Inc., 2022, accessed: July 2022. [Online]. Available: https://github.com/google/syzkaller/

[71] S. Wegne, "Fitbit charge 3 teardown," 2018, accessed: January 2022. [Online]. Available: https://www.techinsights.com/blog/fitbit-charge-3-teardown

[72] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/avatar-framework-support-dynamic-security-analysis-embedded-systems-firmwares

[73] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1099–1114. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[74] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2007–2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou

[75] W. Zhou, Z. Jiang, and L. Guan, "Understanding MPU Usage in Microcontroller-based Systems in the Wild," in *Proceedings 2023 Workshop on Binary Analysis Research*. San Diego, CA, USA: Internet Society, 2023.