CO3: Concolic Co-execution for Firmware

Changming Liu
Northeastern University

Alejandro Mera
Northeastern University

Engin Kirda
Northeastern University

Meng Xu
University of Waterloo

Long Lu
Northeastern University

Abstract

Firmware running on resource-constrained embedded microcontrollers (MCUs) is critical in this IoT era, yet their security is under-analyzed. At the same time, concolic execution has proven to be a successful program analysis technique on conventional workstation platforms. However, porting it to the MCUs faces challenges, such as incomplete and inaccurate emulation of hardware peripherals, reliance on customized hardware, and low execution speed.

CO3 is a firmware-oriented concolic executor attempting to address these limitations. *CO3* runs the firmware concretely on a real MCU to utilize its fidelity. Unlike previous designs, *CO3* gets rid of the slow or proprietary debugging interfaces for synchronization between the MCU and workstation. Instead, *CO3* instruments the firmware source code to strategically report runtime information via a basic serial port to a workstation where symbolic constraints are constructed and solved. We further combine *CO3* with a semi-hosted firmware fuzzing framework to create a hybrid fuzzer (*SHACO*).

The evaluation shows that *CO3* outperforms state-of-the-art (SoTA) firmware-oriented concolic executors by three orders of magnitude while incurring mild memory and runtime overheads. It is also faster than SymCC, a general concolic executor. When evaluated on the existing benchmark, *SHACO* finds all known bugs in a much shorter time. It also found seven bugs from three new firmware samples. All new bugs have been confirmed and patched responsibly.

1 Introduction

In this era of Internet-of-Things (IoT), firmware running on resource-constrained microcontrollers (MCUs) are fundamental components of the cyberspace infrastructure [5]. MCU-based firmware controls essential facilities like power grids, factory machines, and smart homes. The security of the firmware is of paramount importance [29].

Unfortunately, the security situation for firmware is dire. Attacks exploiting vulnerabilities in firmware have raised serious concerns [36, 41, 44]. This is primarily because firmware runs on constrained devices for efficiency and cost reduction purposes. As a result, they do not have access to the security features (e.g., ASLR) shipped with conventional operating systems (e.g., Linux). MCU devices also lack the necessary hardware support (e.g., MMU) on which many security mechanisms depend. Recent research [59] shows that attack mitigation techniques on MCUs are significantly lagging compared to their desktop counterparts. Consequently, detecting bugs in the firmware before deployment is imperative to mitigate these security threats.

On the front of vulnerability detection, concolic execution—a prominent software testing technique—has seen tremendous success and demonstrated great edges over traditional symbolic execution in workstation applications (e.g., desktop and server programs), especially in identifying critical security risks [26, 27, 49, 52, 56, 60]. Different from classic symbolic executors (e.g., [10, 55]) which aim to explore all execution paths at the same time, a concolic executor requires an concrete input to bootstrap [26]. Through maintaining both symbolic and concrete states as the program-under-test executes, a concolic executor gains more flexibility in using different states to explore different parts of the program, taking the advantages of both sides. As an example, a majority of program states can be explored concretely (e.g., by a fuzzer) in a cost-effective manner, while the symbolic engine is only engaged to break through "narrowly guarded" conditions. This forms the logical foundation of hybrid fuzzing [56, 60, 63].

Besides this intrinsic synergy with fuzzing, concolic execution also addresses a scalability limitation in classic symbolic execution, modeling the execution environment (e.g., system calls or even hardware). Concolic execution typically uses taint tracking to distinguish symbolic values from concrete ones at runtime. This allows a concolic executor to observe whether interactions through an environment interface are symbolic. If not, there is no need to symbolically model this interface. In comparison, determining this information statically is intractable for classic symbolic executors.

In firmware testing, recent years have seen a plethora of bug detection techniques tailored towards firmware, as evident

by the extensive list of fuzzers [9, 24, 38, 42, 43, 50, 51, 64] and symbolic executors [4, 13, 15, 20, 28, 40, 61]. Due to the resource-constrained nature of MCUs, performing fuzzing or symbolic execution entirely on the MCU is impractical. As a result, recent works tend to *rehost* the firmware from its native environments (i.e., the MCU) to a much more powerful platform (i.e., a workstation) either through emulation [20, 33] or hardware-in-the-loop [4, 13, 17, 35, 45, 61] (HiL). Emulation inherently does not meet the concolic execution's requirement, which requires an authentic execution environment (e.g., the real hardware), while MCU emulators are rarely complete nor accurate in their modeling [23].

On the other hand, while existing HiL approaches [17, 28, 35] engage the real hardware using debugging interfaces, they have not yet formed into a firmware-oriented concolic executor. Specifically, HiL-based symbolic executors are used as a standalone system instead of being combined with fuzzing. Moreover, they rely on versatile yet complex debugging facilities for synchronization between the MCU and workstation. Unfortunately, established research shows that this debugging interface is often the bottleneck of HiL-based systems [17, 35, 61]. This is because a complex procedure is involved to perform simple functionalities. For example, as illustrated in SURROGATES [35], seven steps are typically involved to perform a simple MCU hardware access. Each step further involves multiple gdb messages. As a result, as low as five hardware accesses can be performed per second. Many works strived to crack this bottleneck. One popular theme of solution is to use expensive or even commercially unavailable FPGA-based debugging interfaces [17, 35]. However, the sheer cost and customization required by such approaches deeply undermine their applicability. Besides, HiL approaches cannot support all hardware features available on a MCU. For example, they require halting the MCU's processor for state synchronization. This violates the real-time requirement that firmware commonly relies on.

Instead of accelerating the debugging process, we took a different route: to avoid the debugging interface completely. We designed a system, *CO3*, to reduce the synchronization cost between MCU and workstation, making *CO3* a practical concolic execution for firmware testing. More specifically, *CO3* faithfully executes firmware on the MCU with real hardware but keeps all hardware events local to the MCU without forwarding them. Instead, *CO3* instruments firmware source code strategically which instructs the firmware to *report and only report* concrete execution states that are relevant to symbolic reasoning to the workstation at runtime. The workstation handles resource-demanding tasks such as building and solving symbolic constraints by receiving firmware states from the MCU instead of emulating the firmware. This design combines key features of MCU and workstation, i.e., full-fidelity concrete execution and powerful symbolic reasoning with low communication cost. *CO3* does not need any debugging interface or customized hardware to bridge MCU and workstation; only a basic serial port is sufficient, warranting good applicability.

We evaluate *CO3* as being three orders of magnitude faster than the state-of-the-art (SoTA) HiL approach. It is even faster than SymCC [49], a SoTA workstation concolic executor. We further experimented with hybrid fuzzing, another benefit of concolic execution, through combining with SHiFT, a SoTA semihosted firmware fuzzer. We named this combination *SHACO*. We compare *SHACO* with various bug detection tools for firmware. *SHACO* outperforms all of them in identifying all known bugs in a much shorter time. *SHACO* also found previously unknown bugs, including three undefined behaviors and four buffer overflows in three actively maintained firmware samples, all confirmed and patched by the developer.

In summary, we make the following contributions:

- We introduce a novel design of concolic executor for firmware, *CO3*, which engages the MCU without resorting to debugging interface. While utilizing MCU's hardware, *CO3* greatly outperforms the traditional HiL-based design, achieving workstation-level performance.
- We combined *CO3* with a semi-hosted firmware fuzzer, SHiFT, to build the first hybrid fuzzer for firmware testing (*SHACO*) which outperformed all SoTA works. Additionally, *SHACO* found seven previously unknown bugs in three new firmware samples.
- Artifacts and source code of *CO3* are available at https://github.com/Lawliar/CO3

2 Background

2.1 Architectural Features of the MCUs

MCUs are self-contained and resource-constrained by design for economical and energy-saving reasons [2]. These design goals make them ubiquitous in critical infrastructure such as medical and industrial devices, aerospace, defense, and automotive [1]. Typical MCUs implement a single address space with designated RAM, FLASH, and peripheral areas. The peripherals' blackbox nature and diversity make them hard and tedious to model [23] in an emulated environment. The core processor interacts with these components in three ways: MMIO, interrupts, and DMA [24]. Thus, an ideal firmware concolic executor should support these three channels without modeling the peripherals.

Previous work [46] categorized embedded systems into three types (I, II, and III) according to their software architecture. This paper focuses on firmware running on Type II (i.e., RTOS-based) and Type III (i.e., bare-metal) MCUs. Type I MCUs support general OSes (e.g., Linux) and have more computation power, making running existing concolic executors (e.g., SymCC) on these systems feasible.

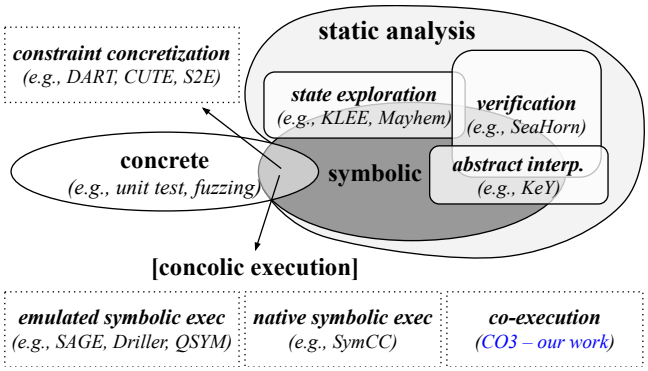

Figure 1: Program analysis techniques that intersect with symbolic and concolic execution and how *CO3* is positioned.

2.2 Symbolic and Concolic Execution

Despite being prominent program analysis techniques, both terms (i.e., symbolic and concolic execution) are overloaded and are sometimes used interchangeably. In this section, we pin down the definition and distinction of these terms for this paper, build a knowledge map on this topic (see Figure 1), and position *CO3* on the map.

The concept of symbolic execution is proposed to differentiate it from concrete execution. In a concrete execution, a program runs on a concrete test case, and a single execution path is explored. In contrast, symbolic execution can simultaneously explore multiple, if not all, paths that a program could take — by marking partial or all inputs as *symbolic*. A symbolic engine follows the program control flow and produces symbolic formulae representing the flow. These symbolic formulae are fed to a solver to answer queries about the program's properties, such as whether the program can take a particular path or reach a specific (buggy) state.

Classic symbolic executors (e.g., KLEE [10], AEG [6], and angr [55]) aim to symbolically track *all* sources of nondeterministic input. This makes them static despite the name "executor". In fact, for classic symbolic execution, its abilities to check property violations on all execution paths create a nice synergy in the domain of software verification and abstract interpretation(e.g., SeaHorn [30], KeY [3]). However, such an exhaustive exploration strategy faces two major challenges: incomplete environment modeling and path explosion. A concolic executor mainly differs from a symbolic one in addressing these two challenges.

A classic symbolic executor needs to soundly model the execution environment to capture the program's interaction with it. This is particularly hard in the MCU as its environment is composed of highly diverse peripherals. Modeling them has been proven to be infeasible [23]. Even worse, for completeness reasons, the modeling is needed even when the interaction with a peripheral rarely involves symbolic input. To ease the manual modeling efforts, DART [26], CUTE [52], and S2E [15] proposed using concrete values, instead of symbolic ones, to interact with the environment. This convenience comes with the expense of potential unsoundness and incompleteness; luckily, the benefit outweighs the cost in practice. Needless to say, concretization requires the availability of environments (e.g., the actual hardware) and faithful interactions with them.

In addressing path explosion, unlike classic symbolic executors that explore a program in a breadth-first manner, a concolic executor uses a depth-first strategy. It collects symbolic constraints along the path dictated by a concrete input and generates new inputs based on that single execution [27, 52]. Doing so enables better synergy with fuzzing [56], arousing extensive research interests [14, 60, 63].

Also, concolic execution is not to be confused with the hybrid execution proposed in Mayhem [11]. Although hybrid execution does use concrete execution to alleviate the workload of symbolic exploration, its main purpose, however, is to approximate path segments instead of being an independent input generator. This makes it less intuitive to combine with fuzzing. And yet, like Driller [56], Mayhem is emulation-based and requires binary rewriting for concrete execution and RPC for communication — features not commonly available and rather limited on the MCU [8, 47].

As highlighted in two concolic executors (i.e., Qsym and SymCC), a major impediment to concolic execution is speed. Qsym [60] shows that symbolic tracking can be accelerated using dynamic binary instrumentation. SymCC [49] further reveals that even dynamic binary instrumentation is slow: the compiler can implant the symbolic tracking logic directly into the binary — enabling symbolic tracking at native speed!

Figure 1 summarizes the flavors of symbolic execution and the evolution of concolic execution. Built upon the power of native symbolic execution enabled by SymCC, *CO3* lifts its reliance on a UNIX-like system for the purpose of placing the concrete execution on the MCU. *CO3* channels concrete execution with symbolic tracking on the workstation through our carefully designed protocol, achieving native execution on both sides (hence the name, co-execution).

2.3 Symbolic Execution for Firmware

Most symbolic executors built for firmware did not improve the symbolic execution itself, e.g., they adopt the same path-exploration, symbolic tracking strategies, etc., as their workstation counterparts. The major efforts in building such a system have been to resolve the hardware dependencies, as they all adopted the *rehosting* method.

As shown in Figure 2, where we summarize all relevant solutions, no matter it is *(a)*, the binary solutions (e.g., Avatar [61], SURROGATES [35], Symbion [28]) where the firmware is rehosted in an emulator or *(b)*, the source code solutions (FIE [20], Inception [17]) where the source code is interpreted in a symbolic engine, they address this problem in

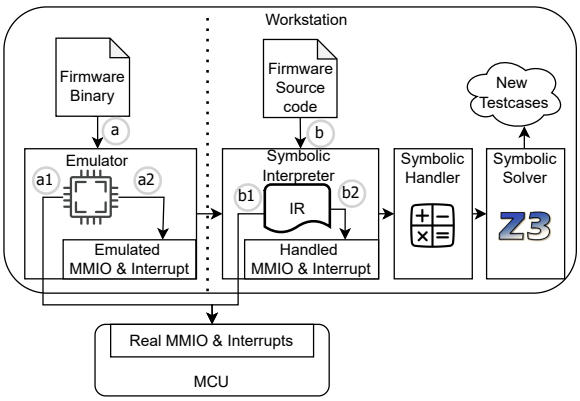

Figure 2: Existing design of symbolic execution for firmware.

the same way. Specifically, they either engage the real hardware through the HiL design (denoted in Figure 2 by *(a1)* and *(b1)*) or emulate the hardware through heuristic-based rules or symbolic values (denoted by *(a2) and* (*b2)*). FIE [20], for example, follows *(b2)*. It symbolically interprets the source code of the firmware and handles its MMIO by symbolizing them while firing interrupts according to a pre-defined specification.

Binary and source code solutions address very different problems. Source code solutions have richer semantics but are dependent on the availability of source code, while binary solutions are, in theory, more universal but have a harder time locating and distinguishing components in MCU's address space. Nonetheless, these works have severe limitations. The HiL approaches require highly customized connectivity interfaces due to outrageous synchronization latency between the MCU and the workstation. For example, SURROGATES [35] and Inception [17] use tailored, commercially unavailable FPGA-based debugging interfaces to connect the MCUs to the workstation. Such expensive customization deeply undermines their applicability. Yet, they are still slow (i.e., slower than KLEE) and fail to support all hardware features (e.g., real-time operation). To use the emulated hardware, the models proposed by the emulation approaches are inaccurate and incomplete. This leads to various problems caused by the lack of fidelity [23]. In contrast, using a better model, or even the real hardware, can prune the symbolic exploration [17] and mitigate state explosion and false positives [43]. In summary, existing solutions are limited in fidelity, speed, or applicability. Hence, these solutions have difficulties finding more, deeper, and real bugs in firmware.

3 Reflection on Hardware-in-the-Loop (HiL)

At the expense of physically engaging the hardware, HiL-based techniques excel over emulation-based approaches in reducing false positives. However, in current HiL designs, hardware is only involved in a very *passive* manner. Specifically, HiL requires a cumbersome procedure to communicate messages and commands with the MCU. For example, to request MMIO data on behalf of the firmware-under-test, the workstation has to: 1) pause the MCU, 2) set a hardware breakpoint at a specific address on the MCU, 3) command to resume the execution, 4) wait for the breakpoint to be hit, and finally, 5) retrieve data from the MCU. This process happens every time a hardware interaction occurs. It relies heavily on debugging protocols and facilities which, even when publicly available, are not intended for speedy testing.

In this paper, we challenge this "intrusive" debugging practice by asking whether we can make the MCU work with the workstation *cooperatively and voluntarily*. Drawing an analogy to hypervisor technology, *CO3* is an experimental solution that transits from full virtualization to paravirtualization for smooth and accelerated analysis on firmware. In the context of concolic execution for firmware, "paravirtualization" means that the firmware running on MCU is fully aware that it is executed concolicly and will proactively report all necessary information for symbolic tracking to the workstation.

This idea is partially inspired by SymCC, which instruments and compiles the target program to implant concolic execution capability into the program itself. The instrumented code allows the program to *actively* taint-track symbolic values and instructions instead of waiting for an interpreter to *passively* single-step it. In *CO3*, we take this concept to the next level by enabling concrete execution on the MCU platform, and channeling it with the symbolic execution on the workstation, realizing native executions on both platforms.

This results in three major benefits:

1. It assures the highest fidelity of all hardware features available on the MCU without having to model them, thanks to the nature of concolic execution.
2. There is no need to forward any hardware event (e.g., MMIO), as they all happen inside the MCU. In doing this, we do not need to host firmware emulation and can directly construct symbolic formulae on the workstation.
3. It takes advantage of the plentiful computing resources on the workstation for heavy symbolic tracking.

However, *CO3* does require source code for instrumentation. For MCU vendors and firmware developers who intend to boost the security assurance of their firmware, source code access is not a limitation. However, we acknowledge that *CO3* is not a solution for blackbox testing for firmware.

Remaining Challenges Although the above concept already shows noticeable advantages over the HiL design (e.g., support all hardware features, no need for firmware emulation and heavy synchronization), two major challenges and questions remain: 1). how does the workstation interact with the MCU and how to minimize the impact of this interaction? 2). how can we symbolize the physical input channels on the MCU due to the lack of a common abstraction layer? The

former is particularly important, as the existing HiL works engage the MCU through a cumbersome debugging protocol which forces them to use prohibitively expensive debugging interfaces. *CO3* needs to come up with a lightweight protocol that can sustain the most commonly available interface—the serial port [43]. In *CO3*, we achieved this via conservative compile-time analysis to extract (1) a skeleton of the symbolic formulae which we termed symbolic value flow graph (SVFG), and (2) as many constants in the formulae as possible.

4 Design

To demonstrate how SVFG works, take Figure 3 as an example. The left side depicts the instrumented firmware running on the MCU in the form of LLVM IR. The right side depicts the symbolic handler on the workstation with the knowledge gained from compile-time analysis of the firmware. Through compile-time static analysis, we already know all symbolic operations to build the symbolic formulae and some values for the operands (i.e., values in green). Reporting these operations and operands is redundant. Instead, the firmware only needs to communicate missing pieces from the MCU to the workstation (i.e., underlined red values), and the firmware should report them at runtime when they are known. Interested readers may refer to Appendix A.1 for a verbose reading of this example.

At *CO3*'s very core is a compiler that can be abstracted into the following interface:

$$\textbf{compile}(src) \rightarrow bin^{*}\,(\text{MCU}) \;+\; SVFGs\,(\text{workstation})$$

Specifically, *CO3* takes in the source code of the firmware and produces two outputs:

- bin^{*}: an executable binary of the firmware for the MCU with instrumented value reporting;
- $SVFGs$: SVFGs on how to build symbolic constraints by consuming reported values from the MCU.

The actual process of a single concolic execution round is distributed across two platforms, as abstracted below:

[on MCU]:

$\textbf{execute}(bin^{*}, input) \rightarrow result \;+\; trace\,(\text{reported on-the-fly})$

[on workstation]

$\textbf{build}(SVFGs, trace) \rightarrow constraints$

$\textbf{solve}(constraints) \rightarrow input$ # send to **execute()** on MCU

Interested readers may also consult Figure 4 for a more complete and graphical overview of this process. The rest of this section focuses on the design details of the SVFG, and symbolizing MCU physical input channels.

4.1 Symbolic Value Flow Graph (SVFG)

In *CO3*, an SVFG is a collection of data-flow slices obtained at compile-time. Each data-flow slice captures how symbolic values flow through a fraction of the firmware execution at runtime. The right side of Figure 3 is a sample SVFG.

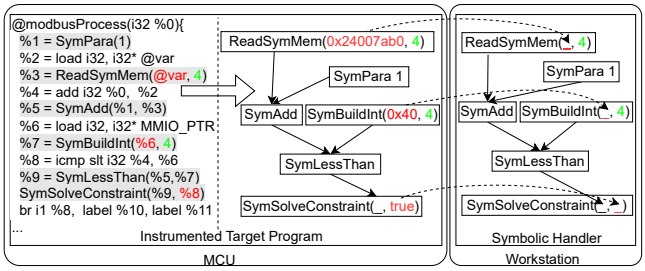

Figure 3: A simplified example of symbolically-instrumented firmware running on the MCU and the SVFG on the workstation. Shaded lines are the instrumentation code in LLVM IR, unshaded lines are the original instructions. Values in green are already known at compile-time. Values in red are unknown until runtime and, thus, need to be reported.

Symbolic value flow stems from three different places, namely, ① constant literals, ② memory reads, and ③ function parameters. These nodes are named *leaf nodes* in an SVFG. Through a series of computations (via *computation nodes* ⊠), these value flows will end up in one of two places: ❶ passing to solver, or ❷ memory write. These two places are known as *root nodes* in an SVFG, and represent how constructed symbolic formulae are ultimately used.

Multiple SVFGs can be connected in two ways: (1) via a memory write then read at the same memory address, or (2) via a function call where the caller passes the value as an argument. On the compiler side, the process of extracting SVFGs is to convert the firmware source code into a static single assignment (SSA) form that encodes the data-flow information. In marking symbolic nodes, the rule of thumb is to treat 1) every memory read (i.e., the `load` instruction in LLVM IR) as a leaf symbolic read, and 2) every memory write (i.e., the `store` instruction) as a root memory write; unless the compiler can decide the values are compile-time constant.

The task on the workstation side is quite straightforward. Information reported by the MCU in real-time instructs which SVFGs to construct together with:

- concrete values for non-symbolic operands in computation and pass-to-solver nodes (e.g., the `true` value in `SymSolveConstraint` in Figure 3).
- memory addresses where the content is symbolic when accessed at runtime to link SVFGs via ❷→② flow. (e.g., address 0x24007ab0 in `ReadSymMem` in Figure 3).
- function calls at runtime to select which SVFGs to build and link SVFGs via function call→③ flow.

The workstation only needs to backtrack the SVFGs from their roots to the leaves to reconstruct the symbolic formula.

4.2 Runtime Operations on MCU

In this part, we focus on how the firmware is instrumented to properly track and report information that the workstation needs to construct symbolic constraints.

① **Constant literals.** It is common to build symbolic formulae containing values already known at compile-time. For this type of leaf node, *CO3* directly embeds their literal values inside the SVFGs stored at the workstation; thus, the MCU does not need to report them.

② **Memory reads and ❷ writes.** MCUs implement a single address space where RAM, FLASH, and MMIO are mapped into different regions that are predetermined per architecture. For example, on ARM Cortex-M, the most popular architecture for MCUs, FLASH is in $[0x00000000, 0x20000000)$, RAM is in $[0x20000000, 0x40000000)$, and MMIO is in $[0x40000000, 0x60000000)$. This allows the firmware to distinguish which memory region its memory access is about by examining the value of the address.

RAM functions like a working memory where the firmware can read and write arbitrary values, including values controllable by user input (i.e., symbolic values) and supplied at runtime by the environment, such as peripherals (i.e., concrete values). As a result, two SVFGs can be connected in two ways:

1. A root node of SVFG_1 writes a *concrete* value to address X and a leaf node on SVFG_2 reads from X.
2. A root node of SVFG_1 writes a *symbolic* value to address X and a leaf node on SVFG_2 reads from X.

Note that case 1) is not interesting to *CO3*. Although it can, the firmware does not need to report the concrete value nor the memory address because they does not involve any symbolic operation. Also this information can be subsumed by a reporting of non-symbolic operands of the computation node. Case 2) on the other hand, is crucial in building the symbolic constraint because this allows two SVFGs to be merged and the continuation of formula building without the need for a symbolic memory model.

In order to track the mapping between memory address and symbolic values, conventional concolic executors use *shadow memory* [49]. The shadow memory essentially stores symbolic values indexed by their corresponding addresses. *CO3* incorporates the same idea, but needs to make one decision: where to host the shadow memory?

Since the MCU is too limited, the expensive shadow memory has to be hosted on the workstation. However, separating the shadow memory from the concrete execution requires the MCU to report the memory address when executing *every* `ReadSymMem` and `WriteSymMem` instruction. In doing so, the workstation is informed with all interactions between the firmware and RAM, and thus can keep its shadow memory up to date as the concrete execution goes on the MCU. However, this approach would incur much traffic since the firmware interacts with RAM intensively. Furthermore, if the user-controlled input is narrowly scoped, most of the reporting would not be interesting to the workstation as they fall into case 1).

Therefore, in *CO3*, we task the MCU to keep track of the *symbolic state* of the RAM. This is done by implanting another *shadow memory* at the MCU side. Instead of storing the real symbolic values of each byte like its workstation counterpart, this shadow memory only stores the symbolic state for them, i.e., each bit in this shadow memory indicates the symbolic state of one byte in the RAM of the MCU. Through the MCU-side shadow memory, we enable the MCU to memorize the symbolic state of every byte on RAM. This allows reporting only for the store and load operations that would affect the shadow memory on the workstation. For example, when *CO3* tries to read from RAM, the MCU can quickly check if this needs to be reported by checking the shadow memory and only report the address when any of the bytes being read is symbolic. This reduces unnecessary traffic significantly.

For ❷ memory writes, the MCU will examine the symbolic state of the value to store and the piece of memory being written. If both are concrete, i.e., writing a concrete value to a concrete memory, no action needs to be taken. However, if any of them is symbolic, the MCU will report to the workstation about the address it is writing to, and update its shadow memory accordingly. The workstation will use this to execute the write operation, and update its shadow as well.

Other memory-mapped regions. For FLASH, due to its read-only nature, no symbolic value will be written to it, and no symbolic value will come from it. Thus, the MCU will always treat its read as *concrete* and do nothing. On the other hand, MMIOs, which the firmware uses to interact with the environment, are complex and hard to model. Thus, *CO3* uses concrete values to interact with the MMIOs to use the real peripheral hardware. In writing to MMIO, the firmware will simply write the concrete runtime values to it; any symbolic taint of that value, if exists, will be lost. The only exception on MMIO handling is the data register (DR) which serves as a source of symbolic input and is discussed in detail in 4.4.

③ **Function Parameters.** Like RAM, the symbolic taints for function parameters and return values must be properly set when executing the call instruction. More specifically, the caller needs to update the symbolic state of the parameters so that they can be correctly referred to in the callee. To that end, we designate another static section of RAM on the MCU to keep track of the symbolic states of the function parameters when executing the call instruction. This is only to properly propagate the symbolic states of the parameters between the caller and the callee; thus, no reporting is needed. The workstation does the same, except it passes the real symbolic values instead of the symbolic states.

X Non-symbolic operands in computation nodes. A major source of reporting is when a computation node involves operands in contrasting symbolic states. To illustrate, consider a `SymAdd` operation that reads its left-hand side (LHS)

operand from RAM and the right-hand side (RHS) operand from MMIO. According to the MCU-side *shadow memory*, the LHS operand is symbolic. And the RHS is concrete. Thus, there is address reporting for the LHS and no address reporting for the RHS. As a result, the workstation will read the symbolic values from its shadow memory for the LHS and mark the RHS as concrete. However, to successfully build this `SymAdd` operation, the concrete value for the RHS also needs to be reported. The instrumented firmware is aware of this situation due to taint tracking. Hence, it will report the concrete value of the RHS to the workstation.

Taint tracking for computation nodes. After establishing the sources of the SVFG, these symbolic values will go through a series of computations. From the MCU's perspective, this process serves merely as the propagation of the symbolic states. Like taint tracking, the result is just a logical disjunction of the taint states of all its operands. Thus, we replace these symbolic operation instructions all with a simple logical disjunction instruction at compile time, reducing the binary size.

Pass-to-solver nodes. This type of node corresponds to a conditional branch that can be toggled to generate new input. In such a case, the MCU will report to the workstation which branch the concrete execution took, and the workstation will negate this formula to pass to the symbolic solver to generate new input that goes to a different branch than the one taken by this concrete execution.

4.3 Complication of the SVFG

Although Figure 3 shows a tree-like structure for SVFG, it is actually a graph because of the existence of ϕ-node in SSA [19]. Code in SSA form uses ϕ-node to choose between different values according to different control flows, and it is common to use ϕ-node on loop boundaries. Regarding the SVFG, a ϕ-node can introduce a back edge to its tree-like structure — making it a graph. For example, depending on whether the control flow comes before the loop (i.e., the first iteration) or inside the loop (i.e., the subsequent iterations), such ϕ-node will take its initial value, which is defined before the loop or the incremented one defined inside the loop. This requires *CO3* to embed control flow information in the SVFG, as we need it to inform which value the ϕ-node chooses on the side of the workstation.

To that end, we group computation nodes by the basic block (BB) they belong to. The naive solution would be for the firmware to report every BB executed. In doing so, we inform all ϕ-nodes on the workstation about which value they should choose, as this can be quickly checked by looking at the immediate preceding BB. Furthermore, the workstation is also informed about the loop, e.g., how many times it has been iterated and which branch it took inside the loop, if any.

Straightforward as this solution might sound, it has a severe drawback: if all operands of a ϕ-node are not symbolic, this reporting is redundant—this is similar to why reporting operands of a `SymAdd` is redundant when both operands are concrete. Reporting those BBs ramps the traffic up but is not interesting to the workstation. Therefore, before reporting it to the workstation, the instrumented firmware needs to check if the BB contains any symbolic operation. To achieve this, *CO3* instruments an instruction in the firmware that aggregates the symbolic states of all nodes within the BB through logical disjunction in compile-time at the end of each BB. The resulting bit indicates if this BB contains any meaningful symbolic operation and should be reported at runtime.

While this fix eliminates the unnecessary traffic, it introduces another problem which we call *symbolic residual*. To illustrate, consider an in-loop BB that contains one read operation from the shadow memory. When it is first executed, this read can be symbolic (i.e., the memory cell contains symbolic value). As a result, such a read and the BB it belongs to will be reported to the workstation. However, when that BB is executed again, the read shifts address, and becomes concrete. Worse still, the other operations within the same BB are also concrete. Neither the read nor the BB would be reported in such a case. Hence, their symbolic states will not be updated on the workstation side. Subsequently, if any formula depends on such a read, it will wrongfully use its symbolic expression that remains on the workstation.

As a result, we also report the BB to the workstation once it switches wholly from symbolic to concrete to eliminate this residual. Note that only the state switch will be reported. If the BB stays concrete, it will not be reported. Although the read is not reported in the previous case, the BB will report a state switch. As a result, the workstation will execute the nodes within the BB again. When it executes the read operation, it will know that the read does not come from a read reporting. Hence, it will know that this read is concrete and mark it as such, keeping it up-to-date. Also, on the MCU side, to memorize the state of the previous execution of every in-loop BB, we allocate one bit per in-loop BB on the function's stack.

After we switch from reporting every BB to only "symbolic" BBs, we cannot inform the ϕ-node on the workstation in the old way, i.e., by checking the immediate preceding BB, as there is no guarantee that either the preceding BB or the BB containing the ϕ-node would be reported. However, following the same idea of reporting only the symbolic BBs, we do not have to report for every ϕ-node, i.e., we only need to report and construct symbolic ones. Thus, what we do instead is to instrument right after each ϕ-node to check which value is chosen and only report when the chosen value is symbolic. Also, the ϕ-node has the same issue of *symbolic residual*, i.e., it switches from symbolic to concrete, but the workstation is unaware. We adopted the same solution as the loop to eliminate its residual.

4.4 MCU Physical Input Channels

As mentioned in Section 2.1, unlike the workstation with well-defined abstracted interfaces (e.g., the POSIX-like API), the MCU uses raw physical channels. MMIO, interrupts, and DMA were identified, summarized, and recognized as physical input channels by previous works [23, 24]. Unlike emulation approaches that use manually crafted heuristics to dynamically identify the input channels, *CO3* can easily locate these channels in the source code based on the insight that the firmware always configures the input channel before using it. To handle these channels, *CO3* introduces a monitor to direct the input from the workstation to the physical channel used by the firmware. This monitor receives input from the workstation, feeds it to the specified physical channel and starts executing the instrumented code. We discuss these three channels in more detail as follows.

Memory-Mapped IOs (MMIOs) The core processor accesses the MMIO region to interact with the peripherals. The data registers (DRs) are the main channel through which raw data flows from peripherals to firmware [24]. To support these DRs, emulation-based fuzzing approaches, such as P2IM, first locate them inside the MMIO region dynamically; then whenever these DRs are read inside the emulator, fuzzer-generated input data is provided.

Following P2IM, *CO3* also directly feeds input data to the DR through the monitor. However, unlike P2IM using pattern matching to identify the DRs, *CO3* automatically locates the DR through a static code scanning and configures the monitor to feed data to it. During online testing, as the workstation sends input to the monitor, the monitor will feed such input to the DR for the firmware to execute upon. Meanwhile, since the workstation also has symbolic memory that needs updating, the firmware must also communicate the DR to the workstation. Specifically, the firmware reports the address and subsequently every read of the DR as it executes. This enables the workstation to feed input data to the DR's address on its end for consumption whenever it receives the message.

Direct Memory Access (DMA) DMA allows data transmission through a DMA controller without the processor's intervention. DICE [42] observes that to use the DMA for receiving inputs, the DMA controller has to be configured with an input source and a destination buffer. After configuration, the DMA controller automatically moves the data from the input source to the destination buffer for consumption.

Following this observation, *CO3* directly symbolizes the destination buffer of the DMA. Specifically, the user first specifies *CO3* to use the DMA channel. Then, *CO3* will automatically locate the destination buffer during symbolic compilation. When the testing starts, the workstation sends input to the monitor; the monitor will feed input data to the configured buffer, symbolize it and start executing the instrumented firmware. To properly update the workstation, the firmware reports the address of the destination buffer. Then, the workstation will know where the input data will be consumed and symbolize it on its end.

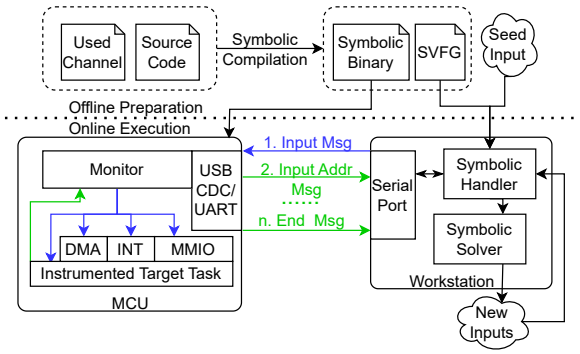

Figure 4: *CO3*'s system overview and communication protocol between the workstation and the MCU in action.

Interrupt Service Routines (ISRs) Firmware follows an interrupt-driven design for efficiency and power-saving purposes. Supporting their service routines is essential [61].

Emulation-based approaches usually fire interrupt according to a pre-defined heuristic due to the inaccessibility of real interrupt-firing information. HiL approaches generally rely on a proxy to broker the triggered interrupt. This turns out to be slow. In *CO3*, since we run firmware directly on the MCU, we only need to instrument the ISRs and let them execute when triggered. The instrumented ISRs behave just like an instrumented function. Meanwhile, since the interrupt can be raised at any time during firmware's execution, it can arrive in the middle of *CO3*'s updating the shadow memory, making it unstable. To solve this problem, *CO3* dynamically disables the interrupt when the firmware enters the *CO3* runtime and re-enables it when the firmware exits it, protecting its integrity.

Designated Buffer Last but not least, to make the *CO3* more flexible, besides supporting all the physical channels mentioned above, we also support the designated buffer as an input source. This is based on our observation that, regardless of what physical channel the firmware uses, it tends to gather them, place them in a buffer, and begin the main processing logic, which is of much testing interest. In contrast, the part of firmware that moves the data from the physical channel to such a buffer is much less interesting. To support such a case, the user just needs to specify which buffer to use before the symbolic compilation. Then, in the testing phase, the monitor will symbolize such a buffer, report its address to the workstation, and start executing the instrumented code. After receiving the address, the workstation will also symbolize the buffer on its end.

Vendor	MCU	CPU	Connectivity	FLASH	RAM
STM32	*F429ZI*	Cortex-M4 @ 180 MHz	UART @ 7.5 Mbauds	2MB	260KB
STM32	*L4R5ZI*	Cortex-M4 @ 120 MHz	USB 2.0 @ 12Mbps	2MB	640KB
STM32	*H743ZI*	Cortex-M7 @ 480 MHz	USB 2.0 @ 12Mbps	2MB	1024KB
NXP	*K66F*	Cortex-M4 @ 180 MHz	USB 2.0 @ 12Mbps	2MB	256KB
Microchip	*SAMD51*	Cortex-M4 @ 120 MHz	USB 2.0 @ 12Mbps	512KB	256KB

Table 1: List of MCU platforms

4.5 MCU-Workstation Communication

To put everything together, we show the overall workflow in Figure 4. First, in the offline phase, the user specifies which input channel the firmware-under-test will use. With this information, *CO3* automatically locates and configures the input channel in the monitor. Meanwhile, *CO3* symbolically instruments the firmware and performs the compile-time analysis to produce the symbolic binary and generate the SVFGs. Then, we feed the SVFG to the workstation and flash the binary on the MCU.

In the online testing phase, the symbolic handler will initiate the *CO3* protocol by sending the seed input. In response, the MCU sends back a message telling how the input is configured. This is followed by all the formulae-building messages, finally ending with an *End* message indicating no more message to send. In parallel with receiving the messages, the workstation constructs the formulae, passing them to the solver to produce new inputs. The newly generated inputs can be fed back to the MCU to continue the symbolic exploration.

5 Implementation

Symbolic Compilation The symbolic compilation is through an LLVM pass based on LLVM-14. Thanks to LLVM's modular design, we only need to instrument at the LLVM IR level, and can then lower the IR into arbitrary architecture supported by LLVM. In this LLVM pass, we first use the user-specified input channel to configure the *CO3* monitor. Then, we conduct use-def chain and loop analysis based on SymCC's instrumentation and output SVFG. We use the DOT language [1] for recording thanks to its flexibility and portability. The whole analysis and SVFG add 2K LoC to the SymCC's compiler. According to our evaluation, the generated SVFGs are typically very small in size, ranging from 60B to 160KB for each function.

MCU Specification As listed in Table 1, we implemented *CO3* on MCUs from STM32 [2], Microchip [3], and NXP [4]. We chose these vendors because they are among the most prominent MCU providers [1].

Note that we use a virtual serial port over USB 2.0 (USB CDC device) or a physical serial port (e.g., UART) as communication channel connecting the MCU and the workstation. Users can easily choose the one that fits their needs. According to a survey in SHiFT [43], these are the two most widely-available communication channels across all surveyed MCUs. We consider this a significant improvement in applicability as the previous HiL works depend on debugging probes that are highly customized or even commercially unavailable.

MCU-side Monitor and Runtime We built the monitor based on FreeRTOS 10.4.2. We chose FreeRTOS because it is arguably the most popular RTOS for Type II devices [5]. Besides, we only use its basic task-scheduling functionalities. Any other RTOS that provides such functionalities is also compatible. For example, we also ported to ChibiOS later in the evaluation.

As shown in Figure 4, the monitor receives inputs from the workstation, feeds them to the specified input channel, and then executes the target program. The instrumented target will then send back messages to the workstation. For this process, we designed 35 messages in total, with each ranging from 2 bytes to 13 bytes in length. We further buffer these messages in a 64-byte data structure on the MCU before sending them to exploit the 64-byte-sized frame that USB 2.0 uses. We developed the runtime functions and monitor from scratch in C. The code comprises only 2.1K LoC in total.

Symbolic Engine *CO3*'s symbolic engine, as shown in Figure 4, renders the SVFGs using the boost library [5]. It then sends and receives messages through the libserialport [6], a cross-platform library for communication between the workstation's symbolic engine and the MCU serial port. Based on the SVFGs and the messages from the MCU, *CO3* calls to the backend of SymCC to construct and solve the symbolic constraints. The symbolic solver is based on Z3. We set the same time-out for the symbolic solver as SymCC to align with the SoTA.

6 Hybrid Fuzz Testing

CO3 explores an firmware program state concolically through generational search [27]. However, for bug-finding purposes, concolic execution alone has two inherent limitations:

- Concolic execution is significantly slower than fuzz testing, especially when exploring easy-to-reach states [56],
- State exploration implies bugs only when there is a bug oracle to decide whether a state is undesired. To illustrate, for conventional desktop/server programs, a bug oracle for out-of-bound (OOB) access can be as simple as page fault (enabled by MMU) or backed by com-

[1] https://graphviz.org/doc/info/lang.html

[2] STM32 MCU Selector, https://www.st.com/content/st_com/en/stm32-mcu-product-selector.html

[3] Microchip Product Selection Tool, https://www.microchip.com/en-us/products/selection-tools

[4] NXP Product Selector, https://www.nxp.com/products/product-selector:PRODUCT-SELECTOR.

[5] Boost C++ Libraries, https://www.boost.org/

[6] libserialport - sigrok, https://sigrok.org/wiki/Libserialport.

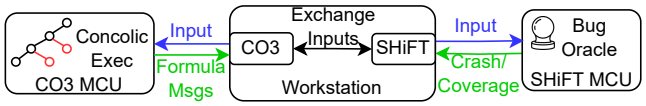

Figure 5: *SHACO*'s overview.

piler instrumentation (e.g., AddressSanitizer [53]). For firmware, however, even OOB might not be observable due to the lack of MMU and kernel support. This lack of observability issue of the MCU has been extensively investigated in wycinwyc [46].

CO3's bug-detection capability can thus be greatly enhanced when combined with a fast fuzzing system that also solves the bug observability issue for the MCU. This is where SHiFT comes in. SHiFT facilities feedback-based fuzz testing (e.g., AFL/AFL++ [25]) natively on the MCU while hosting input mutation on the workstation. It also ported desktop sanitizers to the MCU to address the observability issue. We name this combination *SHACO*, a hybrid fuzzing framework.

As depicted in Figure 5, *SHACO* runs SHiFT on one MCU and *CO3* on another of the same configuration. They both communicate with their workstation component through a serial port. SHiFT is responsible for fast testing and collecting crashes, while *CO3* toggles branches for the inputs deemed interesting by the fuzzer. The input exchange happens on the workstation through a simple file copy. Note that we combine SHiFT and *CO3* in a rather primitive way; as this paper focuses on concolic execution, we leave topics such as a more sophisticated hybrid fuzzer and merging fuzzing and concolic execution to one board for future work.

7 Evaluation

In this section, we evaluate *CO3* to answer the following research questions:

- RQ1: How does *CO3* minimizes the MCU interaction and how does this contribute to the end-to-end performance compared to the SoTA regarding speed and code coverage?
- RQ2: What overhead can a developer expect when using *CO3* for the firmware under development?
- RQ3: How is *SHACO*'s bug detection effectiveness compared to other bug detection works for firmware?

Experiment setup. We perform the evaluation on a workstation with Intel Core i5-7260U CPU, 64 GB of RAM, and Ubuntu 22.04. Details of the MCU used in the evaluation are discussed in Section 5. In fairness, we run all comparison targets using the same workstation, MCU, and communication channel setup. All targets are natively built without patching or modification.

7.1 RQ1: Concolic Execution Performance

Experiment Setup In evaluating RQ1, we select Avatar2 [45], the SoTA multi-target program state transfer framework based on traditional HiL design. Although Avatar2 is not exclusive to testing MCUs, it has built-in support for various debugging probes (e.g., J-Link [7], OpenOCD [8]) commonly used by MCUs. As Avatar2 does not implement fuzz testing nor symbolic execution, we thus compare *CO3* against Symbion [28] which integrates Avatar2 into symbolic execution (i.e., Angr). To be specific, we run both systems against seven real-world firmware samples introduced by P2IM and DICE [42]. These benchmarks are chosen because they have been used extensively by existing works (e.g., Fuzzware, uAFL, and SHIFT). They are *Steering_Control*, *PLC*, *Drone*, *CNC*, *Console* from P2IM; *Midi*, *Modbus* from DICE. For a fair comparison with Symbion, we configured it to perform concolic execution like *CO3* and SymCC. This avoids bias from the path exploration component [9] in angr.

Since *CO3* uses multiple optimizations to minimize the "footprint" of MCU-workstation interaction, we also evaluate how each technique contributes to this reduction. Specifically, *CO3* employs two major techniques: ① replacing a majority of symbolic operations with a logic disjunction instruction on the MCU, and ② MCU-side shadow memory to keep track of the symbolic states of the RAM on the MCU. To see how each technique performs, we have three modes of *CO3*:

- *Report-All*, where the MCU reports every executed instruction without any symbolic states tracking (i.e., disable both ① and ②),
- *Shadowless*, where the MCU performs symbolic state tracking but always treats RAM as symbolic (i.e., disable ②).
- *Full-on*, where both ① and ② are enabled.

Intuitively, *Shadowless* relieves the MCU from having to allocate a piece of RAM as shadow; *Report-ALL* further frees the MCU from keeping any symbolic state. To quantify MCU interactions, we treat the MCU as an IO device and study its impact on the workstation. Specifically, we evaluate 1) how many bytes are transmitted for each run over the wire, and 2) how much time the workstation spends waiting for the MCU to evaluate its impact on the overall performance. These are considered as the "footprint" of an interaction.

Besides the experiment with Symbion, we also compare with SymCC, a SoTA concolic executor. Although SymCC is not designed for firmware, and cannot solve its hardware dependencies like *CO3*, we chose it because both systems follow the compilation-based design. Specifically, SymCC packages symbolic and concrete execution into one binary, while *CO3* further decouples concrete and symbolic execution to heterogeneous platforms and connects them via a serial

[7] https://www.segger.com/downloads/jlink/

[8] https://openocd.org/

[9] Simulation Managers - angr documentation, https://docs.angr.io/en/latest/core-concepts/pathgroups.html#exploration-techniques

port. Comparing with SymCC allows us to assess the value of this decoupling.

In the experiment comparing with SymCC, we use the CGC benchmark with the proof of vulnerability (POV) inputs [48] as seeds. We chose this benchmark because it has been heavily used in SymCC and Qsym for the same evaluation purpose. SymCC and Qsym mentioned many benefits of using CGC programs to evaluate workstation concolic executors. In addition to these benefits, they are also valuable for firmware because of their independence from `libc`, as firmware also typically uses customized `libc` (e.g., `newlib`). We evaluated 5 CGC programs in alphabetical order since these programs need refactoring to run on the MCU and do not represent the real firmware. We consider 5 is enough to understand how *CO3* is compared with SymCC.

Execution Time We measure *CO3*'s execution time between the first *Input Address* message (Figure 4) arriving at the workstation until the last call to the symbolic solver finishes. SymCC measures the time between the start of program execution until the last call to the symbolic solver finishes. Both *CO3* and SymCC use the time measurement module from Qsym. Symbion is slightly different since it relies on the debugger; we measure its time from when the debugger connection is established until all constraints are solved. We continue to loop the generated inputs back as new inputs to the system. We keep running for 24 hours and take the average as results. Although we run all three modes of *CO3* for each experiment, we will focus on *Full-On* mode to compare with SymCC and Symbion while showing data for the other two. We will systematically discuss the differences between different modes in answering RQ2.

In comparing with Symbion, from Figure 6.(2), we can see that without the physical debugging probe, *CO3* outperforms Symbion by three orders of magnitude. The reason for this advantage is three-fold: Firstly, *CO3*'s optimization of traffic. Figure 6.(3) shows that for the tested firmware, Symbion transmits about 150KB per execution, while *CO3* transmits less than 8KB.

Secondly, *CO3* parallelizes MCU-workstation interactions. As mentioned in Section 3, in a traditional debug-based solution (e.g., Symbion), the workstation has to send command, wait for the MCU to take action and send back the result every time a hardware event occurs. The workstation cannot do anything besides waiting in this process. Whereas in *CO3*, the MCU just puts the data on the wire and does not wait for confirmation. This enables the workstation to interact with the MCU in a separate thread. The workstation may process what has been sent by the MCU while waiting for the next message. Figure 6.(6) shows that, in *CO3*, the MCU-workstation interaction has about 1% impact on the end-to-end execution time, resulting in a mere 0.0001s to 0.001s waiting; whereas in Symbion, waiting for MCU has about 23% impact with time ranging from 3s to 7s. From 3-7s to 0.0001-0.001s, *CO3* significantly minimizes the MCU's impact on the workstation.

Lastly, using a debugging protocol to synchronize hardware events is detrimental to performance. This design implicitly requires the firmware to be emulated on the workstation since only the emulator can make use of such hardware events. As a result, the symbolic engine must be placed on top of the emulator. *CO3*, in contrast, instructs the firmware to directly report values for symbolic formulae. This enables us to directly host symbolic engine, removing firmware emulation entirely. As a highlight of the benefits of removing firmware emulation the hardware events forwarding, according to a survey from SHiFT, if used solely to communicate data (e.g., through the `m` and `M` packets[10]), the *GDB* protocol is only about 50% slower than using the raw physical channel [43]. This means that even if we use the *GDB* protocol to implement *CO3*, the result would fetch around 50% throughput of the current *CO3*, which is still significantly faster than the traditional schemes that rely on synchronizing hardware events and firmware emulation.

As for comparing with SymCC, from Figure 6.(1), we can see that *CO3* is even faster than SymCC. This result shows that although *CO3*'s offloading concrete execution onto the MCU incurs extra serial communication overhead and a slowdown on the concrete exeucution, the benefit of this parallelization still outweights the disadvantages.

Conceptual formulation To put the benefits of *CO3*'s design into more perspectives, we formulize all three systems in Figure 7 where we denote A as the cost of *concrete execution and concreteness checking* [49], and B as the cost of *symbolic constraint building and solving*. Then we can formulize SymCC as

$$SymCC = A + B \tag{1}$$

Since *CO3* offloads A to the MCU and incurs serial communication overhead, *CO3* can be formulized as:

$$CO3 = max(xA, B, C) \tag{2}$$

where x is how much a commodity CPU (e.g., Intel-i5) is faster than a MCU (e.g., ARM Cortex-M7) and C is the serial communication overhead. From Eq 2, we can see that, although *CO3* slows down A by x and incurs C, these three components run in parallel, making *CO3* outperform SymCC in most cases. Lastly, we formulize Symbion as

$$Symbion = E(A) + I(B) + C' \tag{3}$$

where $E(A)$ and $I(B)$ represent emulated execution and interpretation-based symbolic execution as compared to native ones and C' denotes the communication cost plus the MCU action time. We can see that, Symbion not only suffers from delayed MCU communication, it also has slower and highly sequential components.

Furthermore, although the execution time of *CO3* shown in this experiment is reasonable, there is still plenty of room for improvement. Interested readers may refer to Appendix A.2

[10] https://sourceware.org/gdb/current/onlinedocs/gdb.html/Packets.html#index-m-packet

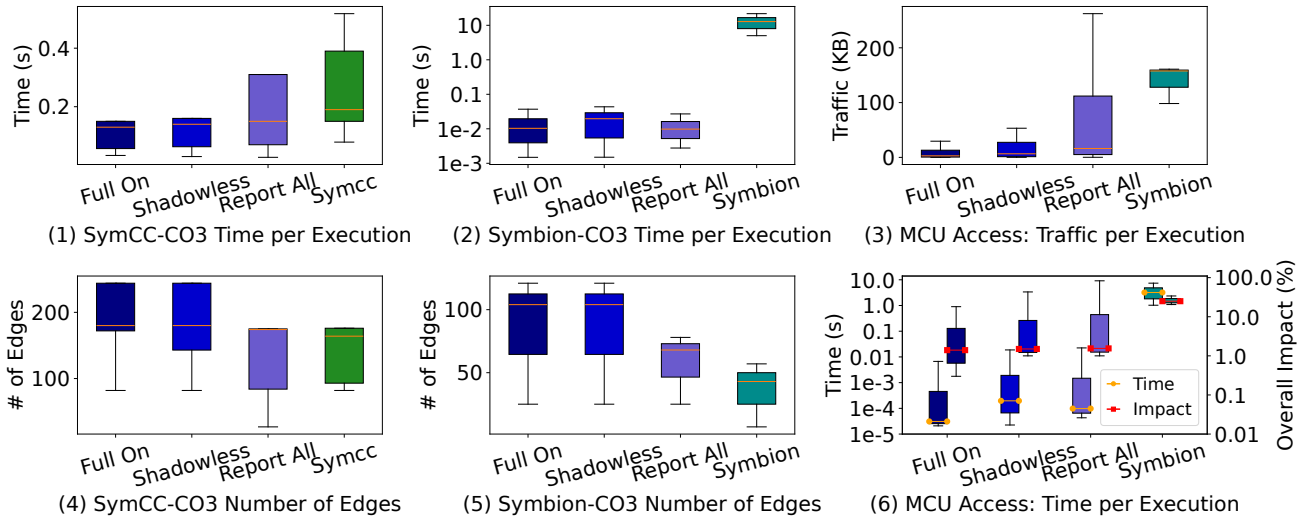

Figure 6: Comparison between *CO3*, SymCC and Symbion for execution speed, edge coverage, and MCU I/O.

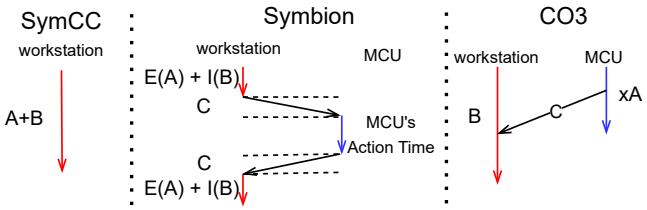

Figure 7: Visualized formulation of SymCC, Symbion and *CO3*

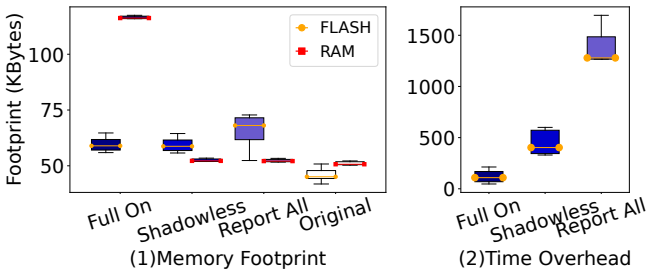

Figure 8: *CO3*'s memory footprint and time overhead compared to the original unmodified firmware.

for more information.

Code Coverage To evaluate code coverage, in the above experiments, we also collect all the generated alternative inputs by all systems. We measure control-flow edges covered by these inputs accumulatively by feeding them to an independent edge-reporting binary. The edge coverage follows AFL's algorithm [11].

As we can see from Figure 6.(4) and 6.(5), for all the tested benchmarks, *CO3* covers more edges. This highlights the efficiency of *CO3* for generating high-quality inputs compared to the SoTA given the same amount of time.

7.2 RQ2: Overheads

This subsection systematically evaluates, in using *CO3*, how much runtime and memory overheads a developer can expect based on all 12 firmware (CGC and P2IM firmware) mentioned above. For each firmware, we prepare the original unmodified firmware and apply three modes of *CO3*. We used STM32-H743, NXP-K66F, and Atmel-SAMD51 from Table 1 as MCU platforms. We focus on *CO3* instead of other systems because these traditional HiL approaches run original firmware on the MCU, theoretically imposing no memory and runtime overhead. Thus, we run the unmodified firmware as the baseline. *CO3* adds multiple components to the firmware. The overheads of such components are vital to evaluate the practicability of *CO3* as the MCU is resource-constrained. Based on the overheads, we also evaluate the trade-off between different modes of *CO3*.

Regarding memory, *CO3* introduces instrumentation, runtime, and the monitor into the firmware. The footprint analysis is presented in Figure 8.(1). From the figure we can see that the original firmware has a median FLASH and RAM footprint of 46.1KB and 52.46KB respectively. *Full-On* mode introduces 12.67KB FLASH overhead, a 27% increase; however, its RAM overhead is 66.1KB, a whopping 125% increase. *Shadowless* mode introduces 12.45KB FLASH overhead, also a 27% increase to FLASH; however, since it does not require one-eighth of the whole RAM as shadow, it only has 1.5KB RAM overhead, i.e., 2.9% increase. Lastly, *Report-All* has a 19.65KB FLASH overhead, a 42% increase. This is because, in this mode, we instrument the original symbolic instructions instead of replacing most of them with a simple

[11] AFL-technical_details.txt https://github.com/google/AFL/blob/master/docs/technical_details.txt#L23

logical disjunction instruction, resulting in a bloated binary. Its RAM overhead is consistent with *Shadowless*, a 1.5KB overhead, as it also does not require shadow memory.

Regarding time overhead, the result is presented in Figure 8.(2). From this figure we can see that *Full-On* mode has a 121x time overhead, while *Shadowless* and *Report-All*'s time overheads are 449x and 1324x. We attributed this to our efforts to minimize the traffic, as serial communication is slower than the MCU's processing speed by far. For example, a simple logic disjunction instruction in *Full-On* can be transmitting 3B in *Report-All*. Note that, for all three modes, the MCU is only responsible for reporting data, it does not perform expensive symbolic operations. We consider *Full-On* and *Shadowless* time overhead mild compared to the 1000x slow-down from a normal concolic execution engine [49].

Considering the overheads and performances of all three modes, although *Full-On* has a 66.1KB RAM overhead, all three modes of *CO3* fit into the three MCU platforms that we experimented on. This is partly because we use relatively high-end MCUs as experimental platforms. In practice, we do foresee cases where the one-eighth RAM overhead from the shadow memory can be expensive. For such a case, we recommend *Shadowless* for its mere 1.5KB RAM overhead.

In conclusion: *Full-On* mode has the best speed and edge coverage performances but its RAM overhead can be prohibitive making it not applicable to all MCUs. *Shadowless* has slightly worse speed and edge coverage performance compared to *Full-On*, but its memory overhead makes it generally applicable if a 449x time overhead on the MCU is tolerable. *Report-All* not only has much worse speed and coverage performance compared to the first two, but its memory and time overheads also make it less desirable.

7.3 RQ3: Bug-Detection Capability

In RQ3, we evaluate the bug-detection capability of *SHACO*. The SoTAs in bug detection for firmware are primarily emulation-based. We select Fuzzware [51], P2IM, and DICE as targets systems. Fuzzware and P2IM are chosen for their popularity and bug-finding capability; DICE is chosen for its unique ability to support the DMA feature. Since *SHACO* combines *CO3* with SHiFT, we also evaluate against SHiFT alone to evaluate what *CO3* contributes in addition to SHiFT.

Benchmark We evaluate the bug detection capabilities of the selected targets from two different angles: 1. the ability to detect *known, documented bugs*, and 2. the ability to detect *unknown, new bugs*. For the former, due to the lack of a well-known benchmark (e.g., LAVA [21]) for MCUs, we chose the firmware samples with well-documented bugs that were widely used in the literature. P2IM and DICE, the two seminal emulation-based approaches, suggested such samples with detailed bug reports; these buggy firmware samples have also been used by other works (e.g., uAFL, Fuzzware). Besides, SHiFT provides firmware samples with bugs that emulation-based approaches cannot find [43]. Thus, as Table 2 shows, we select *PLC (4 bugs)* from P2IM, *Modbus (3 bugs)* and *Midi (2 bugs)* from DICE, *Shelly Dimmer (3 bugs)* and *Synthetic (11 bugs)* from SHiFT to make a benchmark. *Modbus* and *Midi* use DMA as the input channel which Fuzzware does not support out-of-the-box. For a fair comparison, we manually identify the static DMA buffer from the binary with debugging information and annotate in fuzzware's config file [12]. We run *CO3* and related works for 24 hours, manually triage all the crashing inputs (**UC** row of Table 2), and measure how long they took to identify **all** the known bugs (**TP** row of Table 2) of each firmware to evaluate their bug-detection capabilities.

In addition to the firmware with known bugs, *SHACO* additionally found new bugs in other firmware. As shown in the *SHACO* column of Table 2, these are: *CANopen*: an open-source firmware of the controller area network (CAN), *Stepper*: a firmware that drives a stepper machine, and *bldc*: a firmware for real-world motor controller. In *CANopen*, *SHACO* finds three undefined behaviors. In *Stepper*, *SHACO* finds two buffer overflows (BOFs). In *bldc*, *SHACO* finds two BOFs. These bugs have all been confirmed and patched by the developers. We also run related works on these firmware to test their ability to find unknown bugs via manually triaging their crashing inputs.

Detecting All Known Bugs With as simple a strategy as exchanging inputs between SHiFT and *CO3*, the time taken to detect all the bugs in the benchmark is reduced significantly. This confirms the effectiveness of *SHACO* in detecting bugs in real-world firmware. In some cases, the improvement is insignificant, such as the 1.9x speedup in *Modbus*. This is due to *Modbus*'s small code base and search space, e.g., only 230 LoC. For such small firmware, fuzzing is usually enough. In *Midi*, *SHACO* is almost identical to SHiFT. This is because *Midi* employs a state-machine-based design. As a result, the execution of an input is dependent on the previously executed inputs. In such a case, the input generated by concolic execution can easily be rendered invalid due to the change of state. We refer interested readers to papers (e.g., SGFuzz [7]) addressing this issue.

However, this is just to compare *SHACO* with SHiFT, as these two approaches both run natively on the MCU and utilize real hardware. The contrast is much sharper when comparing *SHACO* against emulation-based approaches. As shown in Table 2, *SHACO* generally outperforms them by two to three orders of magnitude in terms of bug-detection capability. The emulation approaches not only failed to boot (marked with **NB**) due to the incorrect model (e.g., *Shelly Dimmer* for P2IM) when they did boot, their inaccurate MMIO interaction leads to unauthentic program state (e.g., Fuzzware's MMIO

[12] https://github.com/fuzzware-fuzzer/fuzzware-emulator/blob/4ee7228ceda140cf2e0d7575ad25dcb7dfa142da/README_config.yml#L67

Ref	#	Firmware	OS	MCU	*SHACO*				SHiFT					P2IM/DICE					Fuzzware				
					Time(s)	UC	TP	FP	Time(s)	SUF	UC	TP	FP	Time(s)	SUF	UC	TP	FP	Time(s)	SUF	UC	TP	FP
P2IM [24]	1	*PLC*	**F**	h743	38	8	4	0	165	4.3x	8	4	0	3873	101.9x	183	4	2	73980	1946x	30	4	2
DICE [42]	2	*Modbus*	**F**	h743	20	4	3	0	38	1.9x	4	3	0	29881	1494x	71	3	2	**I**	**n/a**	25	0	1
	3	*Midi*	**F**	h743	126	20	2	0	129	1.02x	20	2	0	25413	201x	4	2	0	**I**	**n/a**	104	0	2
SHiFT [43]	4	*Synthetic*	**F**	h743	26	20	11	0	340	13.1x	23	11	0	**I**	**n/a**	8	3	1	**I**	**n/a**	486	0	10
	5	*Shelly Dimmer*	**F**	h743	40	6	3	0	262	6.5x	7	3	0	**NB**	**n/a**	0	0	0	**I**	**n/a**	1496	0	1
SHACO	6	*CANopen*	**F**	l4r5	164	7	3	0	525	3.2x	8	3	0	**NB**	**n/a**	0	0	0	**I**	**n/a**	0	0	0
	7	*Stepper*	**F**	l4r5	187	7	2	0	691	3.7x	7	2	0	**NB**	**n/a**	0	0	0	**I**	**n/a**	2355	1	3
	8	*Bldc*	**C**	f429	376	4	2	0	2068	5.5x	4	2	0	**NB**	**n/a**	0	0	0	**NB**	**n/a**	0	0	0

Table 2: Fuzz testing campaigns of *SHACO* compared to other SoTA. **Time**: How much time it takes for this work to identify all the known bugs. **F**: FreeRTOS. **C**: ChibiOS. **SUF**: *SHACO*'s speedup factor compared to this work. **UC**: # of Unique Crashes. **TP**: # of True Positive bugs. **FP**: # of False Positive bugs. **I**: Incomplete detection, i.e., it does not detect all bugs in 24h. **NB**: No Bootstrap, i.e., the firmware fails to boot.

model returns practically impossible values). This prevents them from finding all the true bugs (marked with **I**). Adding up to the vast false positives, the situation even worsens considering the bug observability issue (i.e., the triggered bugs usually take a long time to observe) they have.

Detecting New Bugs Due to the hardware and software dependency of the newly introduced firmware, we ported *SHACO* to two new MCUs (i.e., stm32l4r5 and stm32f429) and one new RTOS (i.e., ChibiOS). P2IM cannot boot due to poor emulator support for these MCUs [13] [14]. Fuzzware avoids this issue by using Unicorn instead of QEMU as the emulator; however, it still fails to support the FPU [15], which *bldc* configures in assembly.

After the fuzzing campaign, the selected systems exhibit similar behaviors as in the known bugs detection experiment (i.e., *SHACO* outperforms SHiFT three to five times while Fuzzware is trapped by the unauthentic program states and fail to detect the same bugs).

Case Study In *Stepper*, for example, we found an undefined behavior (UB) bug. This UB is inside a callback function triggered from a multiplexed timer's interrupt service routine. Since *CO3* has access to the real timer's hardware, the timer just normally incremented and triggered different multiplexed callback functions as expected. In both P2IM and Fuzzware, however, since they are at the NVIC's level which triggers each interrupt service routine in a round-robin fashion, they are not aware of this multiplexing. As a result, in all of their executions, they failed to execute this callback function.

Also, in *CANopen*, a buffer overflow where a hard-coded offset is indexed into the variable length input buffer was identified. However, in this firmware, the returned value of `HAL_GetTick()` was used as the divisor. In the real-world setting, such a function would never return zero. However, both P2IM and Fuzzware uses a simple incrementing strategy to model the Systick, which starts from zero. This crashed the firmware early on and prevented it from exploring deeper.

8 Discussion

Support for Other Architectures We demonstrate the compatibility of *CO3* with the popular ARM architecture. However, MCUs are well-known for the diverse architectures they use. Hence, supporting as many other different architectures as possible would be desirable. In this regard, similar to what the rehosting works do, i.e., using QEMU's ability to lift various architectures to its own intermediate representation [16], *CO3* relies on LLVM's modular design to lower the LLVM IR into a wide range of architectures supported by LLVM [17]. There is one small exception: *CO3* also needs to dynamically disable/re-enable interrupts to protect its runtime's integrity. This is done by architecture-specific inline assembly code (e.g., `cpsid if` in ARM). Fortunately, such code is usually very small and lightweight by design. Thus, it does not harm the portability of *CO3*.

Support for Other RTOSes In the evaluation, we ported *CO3* to FreeRTOS and ChibiOS. However, compared to the relatively small number of popular desktop OSes, embedded RTOSes are much more diverse. It would be desirable to support these RTOSes too.

Since *CO3* does not have any RTOS-specific reliance, it is viable to support any RTOS or even desktop OS. That is, regardless of what type of interface the RTOS provides (e.g., POSIX-compliant like NuttX, CMSIS-compliant like Zephyr, or even the ones not compliant with any standard like FreeRTOS kernel), they all have APIs that accomplish basic OS primitives. Porting to these RTOSes can be as simple as replacing the APIs with the ones that perform the same OS primitive. For *CO3*, these primitives are task creation/deletion and semaphore. In other words, as long as these primitives are implemented, *CO3* can be RTOS-less (i.e., baremetal).

[13] https://github.com/RiS3-Lab/p2im/blob/master/qemu/src/qemu.git/hw/arm/stm32-mcus.c

[14] https://www.qemu.org/docs/master/system/arm/stm32.html

[15] https://github.com/fuzzware-fuzzer/fuzzware#floating-point-unit

[16] Documentation Platforms - QEMU, https://wiki.qemu.org/Documentation/Platforms

[17] LLVM Architecture & Platform Information for Compiler Writers, https://llvm.org/docs/CompilerWriterInfo.html

Meanwhile, *CO3* utilizes USB-CDC or UART for data transmission. Different MCU vendors have different hardware and driver implementations for the two peripherals, however, they all follow the USB and UART standard, and they are among the most universal peripherals across all vendors [43]. To use these peripherals for *CO3*, one can simply follow the official source code examples provided by the vendors [18].

Future Works *CO3* makes it much more efficient to symbolically and authentically explore the firmware on the MCU. Going forward, two directions can be further explored to utilize this capability.

The first one is to construct symbolic formulae under *CO3*'s framework not just for toggling the branches but also for bug detection. Many works [18,39,58] tried to symbolically model various types of bugs and pass to the solver to see if the bug-triggering condition can be met. *CO3* has inherent synergy with these works. The second one is to use *CO3* purely as a state explorer. With methods such as generational search [27], *CO3* makes it much more efficient to authentically reach certain points of interest than the traditional HiL works (e.g., Avatar2). After reaching that point, the program state can be transferred to the emulator for more dynamic analysis.

9 Related Work

CO3 improves the SoTA symbolic execution for firmware through the inspiration of recent research advancements in symbolic execution and firmware rehosting.

Symbolic Execution Symbolic execution, as discussed in detail in Section 2.2, has seen success in various aspects of bug detection [12,32,37,39,58,62] and program testing [14,56,60] for desktop programs. *CO3* builds upon the compilation-based design proposed in SymCC [49] and pushes it forward by applying it to the firmware through our co-execution design.

Firmware Rehosting Firmware rehosting is the technique of migrating firmware from its native environment to a usually more powerful environment for better visibility and more computing resources. They can be categorized into 1). pure emulation either through symbolic abstraction [31, 33, 51, 54] which uses symbolic values to model the hardware accesses, or heuristic-based methods [16, 24, 42, 64, 65] which uses heuristic-based rules for the same purpose. 2). HiL [22, 34, 35, 45, 50, 57, 61] where the emulator engages the MCU to utilize its real hardware.

The most related work of *CO3* in this line of research is SHiFT [43], as they both follow the *semihosting* philosophy. *Semihosting* emphasizes that the MCU should actively invoke the resources from the workstation instead of the other way around, which is followed by most HiL-based works. Thus, SHiFT runs most of the fuzzing infrastructure on the MCU and actively consumes the input mutation from the workstation. Similarly, *CO3* runs concrete execution on the MCU and actively consumes the symbolic tracking from the workstation. SHiFT's fuzzing and *CO3*'s concolic execution generate an excellent synergy, which results in *SHACO*.

[18] https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/STM32F429I-Discovery/Examples/UART/UART_TwoBoards_ComIT

10 Conclusion

This paper presents *CO3*, a novel concolic executor for firmware. *CO3* features high hardware fidelity, applicability, and performance. Built upon the compilation-based idea from SymCC, *CO3* places the concrete execution on the MCU and channels it with the symbolic execution on the workstation.

Evaluated against the SoTA concolic executors for both the MCUs and the workstations, *CO3* achieved workstation-level performance. In combination with a fuzz testing system, *SHACO* outperformed emulation-based bug detection approaches and found seven new bugs.

11 Acknowledgement

The authors would like to thank the anonymous reviewers and our shepherd for their insightful comments. This project was partially-supported by the National Science Foundation (Grant#: 2031390, "Rethinking Fuzzing for Security"). We would also like to thank the DAC Convoy Security project for its support. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Microcontroller Market Size, Share & Trends Report, 2030. URL: https://www.grandviewresearch.com/industry-analysis/microcontroller-market.

[2] Microcontroller, May 2023. URL: https://en.wikipedia.org/w/index.php?title=Microcontroller.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification–the KeY book. *Lecture notes in computer science*, 10001, 2016.

[4] Chengwei Ai, Weiyu Dong, and Zicong Gao. A Novel Concolic Execution Approach on Embedded Device. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, ICCSP 2020. Association for Computing Machinery, January 2020.

[5] Aspencore. 2019 Embedded Markets Study, March 2019. URL: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[7] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *The 45th International Conference on Software Engineering (ICSE'23)*, May 2023.

[8] Christiaan Banister. Low overhead remote procedure call system for saturn dsp. Master's thesis, EECS Department, University of California, Berkeley, May 2022.

[9] Matthias Börsig, Sven Nitzsche, Max Eisele, Roland Gröll, Jürgen Becker, and Ingmar Baumgart. Fuzzing Framework for ESP32 Microcontrollers. In *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*, December 2020. ISSN: 2157-4774.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08. USENIX Association, December 2008.

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[12] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017.

[13] Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu. Test generation for embedded executables via concolic execution in a real environment. *IEEE Transactions on Reliability*, 64(1):284–296, 2015.

[14] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1), March 2011.

[16] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[17] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[18] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. *ACM SIGPLAN Notices*, 48(4), 2013.

[19] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[20] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *22nd USENIX Security Symposium*, 2013.

[21] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2016.

[22] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing Embedded Systems using Debug Interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, July 2023.

[23] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, May 2021.

[24] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05. Association for Computing Machinery, June 2005.

[27] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3), March 2012.

[28] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*, June 2020.

[29] Barr Group. Embedded Systems Market Surveys, June 2016. URL: https://barrgroup.com/embedded-systems/market-surveys.

[30] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, Lecture Notes in Computer Science. Springer International, 2015.

[31] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin 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 '17. ACM, October 2017.

[32] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, October 2020.

[33] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[34] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, ASIA CCS '14. Association for Computing Machinery, June 2014.

[35] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[36] Abraham Peedikayil Kuruvila, Ioannis Zografopoulos, Kanad Basu, and Charalambos Konstantinou. Hardware-assisted detection of firmware attacks in inverter-based cyberphysical microgrids. *International Journal of Electrical Power & Energy Systems*, 132, 2021.

[37] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution. In *Proceedings of the Web Conference 2021*. ACM, April 2021.

[38] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. uAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th international conference on software engineering*, ICSE '22, 2022.

[39] Changming Liu, Yaohui Chen, and Long Lu. KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel. In *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[40] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. Mousse: a system for selective symbolic execution of programs with untamed environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. Association for Computing Machinery, 2020.

[41] Federico Maggi, Marcello Pogliani, and P Milano. Attacks on Smart Manufacturing Systems. *Trend Micro Research: Shibuya, Japan*, 2020.

[42] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021.

[43] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[44] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91), 2015.

[45] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A Multi-Target Orchestration Platform. In *Proceedings 2018 Workshop on Binary Analysis Research*. Internet Society, 2018.

[46] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018.

[47] JinSeok Oh, Sungyu Kim, Eunji Jeong, and Soo-Mook Moon. OS-less dynamic binary instrumentation for embedded firmware. In *2015 IEEE Symposium in Low-Power and High-Speed Chips*, April 2015.

[48] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, December 2019.

[49] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[50] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium*, 2020.

[51] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise mmio Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium*, 2022.

[52] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5), September 2005.

[53] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, June 2012.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[56] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[57] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[58] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[59] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building Embedded Systems Like It's 1996. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, March 2022.

[60] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, 2018.

[61] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings 2014 Network and Distributed System Security Symposium*, 2014.

[62] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul Yu. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[63] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[64] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[65] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22. Association for Computing Machinery, November 2022.

A Appendix

A.1 Verbose interpretation of the example

In Figure 3, based on compile-time analysis, *CO3* can statically determine that this symbolic formula reads 4 bytes from RAM; it then adds the read value to the parameter of the function and compares the result with a 4-byte integer from MMIO (as the firmware only concretely interacts with MMIO). The comparison result is then used as a branch condition. In order to build this formula, instead of reporting every executed instruction and its operands, the firmware running on the MCU only needs to fill in the missing pieces, which is known at runtime. Upon receiving the missing pieces, the workstation can construct the symbolic formula and call the symbolic solver. Otherwise, if the MCU does not send any information, this would mean that this formula does not involve any symbolic value. Thus, it will not be built.

A.2 Further Optimizations

Symbolic backends such as Qsym employs instruction pruning (i.e., skipping symbolic construction for repeatedly-executed instructions). Although it is viable to do this at the MCU side to further reduce the USB traffic – which warrants even better performance – doing so would tightly couple the MCU runtime with the symbolic backend on the workstation, making the runtime much less generic. Since the end-to-end performance was evaluated to be satisfactory, we chose not to sacrifice simplicity and generality for performance gains.

Similarly, there are also engineering practices that we can adopt to improve the performance further. For example, we can parallelize the construction of independent symbolic variables (i.e., symbolic variables that have no dependency on each other). We can also use dual buffers on the MCU side for sending messages so that the firmware execution will be less affected by the USB transmission. We chose not to implement these for the same reason of simplicity.