# Discovery and Identification of Memory Corruption Vulnerabilities on Bare-Metal Embedded Devices

Majid Salehi, Luca Degani, Marco Roveri, Danny Hughes, and Bruno Crispo, *Senior Member, IEEE*

**Abstract**—Memory corruption vulnerabilities remain a prevalent threat on low-cost bare-metal devices. Fuzzing is a popular technique for automatically discovering such vulnerabilities. However, bare-metal devices lack even basic security mechanisms such as Memory Management Unit. Consequently, fuzzing approaches encounter silent memory corruptions with no visible effects, making even discovery difficult. Once discovered, it is also essential to identify the type of observed vulnerability for applying mitigation. Both discovery and identification remain open challenges in the case of fuzzing firmware binaries. This article addresses these problems by proposing an automated instrumentation technique that allows the observation of memory corruption vulnerabilities that are otherwise not observable and facilitates the automated identification of the observed vulnerability. Additionally, we surveyed state-of-the-art IoT fuzzers and analyzed their experimental methodologies. We found that existing approaches have fundamental problems that lead to incorrect or misleading results. To evaluate the effectiveness of IoT fuzzers, it is essential to determine the range and type of vulnerabilities that these fuzzers can discover. Thus, we propose the first ground-truth benchmark suite for IoT fuzzers that enables accurate and consistent evaluation of their vulnerability-finding performance. Our instrumentation framework's efficacy and efficiency in combination with state-of-the-art IoT fuzzers are assessed using the proposed benchmark.

**Index Terms**—IoT security, fuzzing, benchmark, memory corruption vulnerability

## 1 Introduction

The use of Internet of Things (IoT) technologies has been increasing in many safety-critical settings, such as industrial control systems, automotive, unmanned aerial vehicles (UAVs), implantable medical devices, etc. Recently, Ericsson [1] predicted that the number of IoT devices will increase to around 19 billion worldwide by 2022. As IoT applications and scenarios are getting more mature and pervasive, they are also becoming a target of malicious and criminal activities.

A significant portion of deployed IoT devices use low cost bare-metal devices, called so because without an operating system. Such devices execute a single binary image (i.e., firmware) in a privileged mode with direct access to the processor and peripherals. For the reason of efficiency, C and C++ are the most popular languages used for developing those firmware. However, such programming languages are neither type-safe nor memory-safe, and memory corruption vulnerabilities such as buffer overflows remain a prevalent threat on such platforms.

Furthermore, compromising these devices is not confined to the device itself, but it can be used to gain access to higher level systems, as recently shown by Google's P0 [2] that exploiting Broadcom's WiFi SoC in mobile devices enabled adversaries to gain control over the main application processor of the mobile device. As a result, it is crucial to discover vulnerabilities also in bare-metal firmware.

Fuzz-testing or fuzzing is a popular technique for automatically discovering vulnerabilities in applications. The main idea of fuzzing is to feed a randomly/guided generated inputs to the target under test in order to trigger bugs through crashes or other observable behaviors. Among all fuzzing techniques, greybox fuzzing is considered state-of-the-art in both industry and academia due to its applicability, lightweight instrumentation and fast coverage feedback, enabling it to reveal thousands of vulnerabilities in real-world applications.

Fuzzers developed for general purpose computers enjoy a wide number of deployed mechanisms for discovery and identification of memory corruption vulnerabilities such as segmentation faults and memory sanitizers. On the contrary, fuzzers for bare-metal devices encounter some obstacles that make it extremely challenging to apply traditional fuzzing mechanisms on firmware directly.

First, as pointed out by Muench *et al.* [3], memory corruption vulnerabilities are less likely to crash the bare-metal firmware than general purpose computers, causing fuzzing techniques to miss some vulnerabilities after encountering and triggering them. Muench *et al.* proposed six simple

- *Majid Salehi and Danny Hughes are with imec-DistriNet, KU Leuven, 3001 Leuven, Belgium. E-mail: {Majid.Salehi, Danny.Hughes}@cs.kuleuven.be.*
- *Luca Degani is with the University of Trento, 38122 Trento, Italy, and also with the Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, 20090 Pisa, Italy. E-mail: Luca.Degani@unitn.it.*
- *Marco Roveri and Bruno Crispo are with the University of Trento, 38122 Trento, Italy. E-mail: {Marco.Roveri, Bruno.Crispo}@unitn.it.*

*Manuscript received 26 February 2021; revised 21 January 2022; accepted 24 January 2022. Date of publication 8 February 2022; date of current version 14 March 2023.*
*This work was supported in part by Research Fund KU Leuven, and in part by Flemish Research Programme Cybersecurity. The work of Bruno Crispo was supported in part by EU H2020-SU-ICT-03-2018 under Grant 830929 CyberSec4Europe.*
*(Corresponding author: Majid Salehi.)*
*Digital Object Identifier no. 10.1109/TDSC.2022.3149371*

heuristics for embedded firmware and integrated them with a blackbox fuzzer in order to make memory corruptions observable. However, these heuristics suffer from their reliance on a set of information that can be obtained only by applying tedious work of advanced static analysis and reverse engineering techniques on firmware. Second, once a number of crashes is found, it is necessary to identify the location and root cause of each crash for fixing the underlying bug. However, crash triaging remains a manual, time-intensive endeavor for IoT fuzzing. Third, the ability to instrument binary images is an essential requirement for detection and identification of memory corruption vulnerabilities. Indeed, instrumentation tools can be utilized for combining memory sanitization methods with fuzzers in order to check every memory access and track (de)allocations for every memory object. Unfortunately, binary instrumentation tools are rarely available for bare-metal devices. Fourth, evaluating fuzzing mechanisms proposed for IoT devices is very challenging owing to the randomness of the process (i.e., each fuzzing run on an application may produce different results than the last due to the use of randomness) and domain specialization (e.g., a fuzzing mechanism may only work for a certain type of vulnerability or in a certain environment). We surveyed and assessed the experimental evaluations conducted by recent research literature and found problems in every one of them. For example, most of the IoT fuzzers counted the number of crashing inputs discovered in order to evaluate vulnerability-finding performance. Nevertheless, different inputs could cause crashes by triggering the same vulnerability, leading to misleading or incorrect conclusions. Some fuzzers employ heuristic-based approaches [4] with the aim of de-duplicating inputs that trigger the same vulnerability, obtaining a ”unique” input for that vulnerability. However, heuristics are ineffective in identifying unique vulnerabilities [5]. In other words, disambiguating crashing inputs and correctly counting the number of discovered vulnerabilities is limited by the lack of a benchmark suite with ground truth for IoT fuzzers.

In this paper, we present an automatic, static binary instrumentation framework designed specifically for discovering and identifying memory-corruption flaws in bare-metal firmware.[1] A unique property of our proposed framework is that it runs a sanitizer-guided instrumentation to embed a given memory safety policy; any violation to the policy triggers an observable warning and causes the firmware to crash. Therefore, running an IoT fuzzer on the firmware being sanitized discover previously undiscovered silent memory corruptions. Furthermore, our framework utilizes the information collected by monitoring all memory reads and writes during firmware execution in a chronological order to identify crashing cause and location. The proposed analysis method not only identifies the actual origin of a crash, but also provides context information on the erroneous behavior that characterizes the crash by the type of potential memory vulnerability (e.g., buffer overflow and use-after-free). In addition, we introduce a benchmark with ground truth to address pressing challenges and limitations in evaluating IoT fuzzers. The benchmark includes bare-metal firmware ranging from cameras to industrial control systems featuring several connectivity protocols and rich interactions with peripherals. In summary, the main contributions of the paper are as follows:

- We present a novel static binary instrumentation technique that allows discovery and identification of memory corruption vulnerabilities on bare-metal firmware.
- We have implemented the technique in a full-featured framework for the ARM architecture which is one of the most widely used IoT device architectures.
- We generated the first realistic benchmark suite containing 18 representative firmware, for evaluating the capability of IoT fuzzers in detecting memory corruption vulnerabilities. The benchmark suite along with the framework implementation are open sourced and available to the research community at https://github.com/pwnforce/uSBS.
- We evaluated the effectiveness of the proposed instrumentation technique using the benchmark suite and demonstrated that it allows automatic discovery and identification of memory corruption vulnerabilities otherwise not detected by current IoT fuzzers. Furthermore, the results also show the efficiency of instrumented binaries in practice with an average runtime overhead of about 50% which is an acceptable overhead since they are only incurred at the testing time not the actual deployed firmware on the field. Also, our framework only takes a few seconds (i.e., in average 334 seconds) to instrument benchmark firmware.

## 2 Background

In this section, we provide background information on memory corruption vulnerabilities and fuzzing as an approach to discover them, and discuss some limitations related to the architecture of bare-metal devices that motivate the need to extend and refine the fuzzing approach on such architectures.

### 2.1 Memory Corruptions and Fuzzing

Low-level systems software such as embedded devices firmware is typically written in the C or C++ programming languages due to the fact that they are efficient and capable to fully control the underlying hardware. In such languages, programmers must ensure that every memory access is valid, that no situation leads to the de-referencing of invalid pointers. As a matter of fact, programmers frequently fail to meet these responsibilities and cause memory vulnerabilities that can be exploited by an adversary to alter the software behavior or even taking full control over the software stack.

As a result, it is essential for security analysts to discover these memory vulnerabilities and fix them before adversaries. Unlike source code analysis and reverse engineering techniques, fuzzing has been proved as one of the most effective and widely used software security testing methodologies for

1. This work is a significant extension of a paper [6] published in 2020 at the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID). We proceed further in the description without relying on any prior knowledge of the published paper.

TABLE 1
Hardware Protection Mechanisms Supported by Representative Core Families

| Core Family | | Hardware Protection Mechanism | | |
|---|---|---|---|---|
| | | MPU | MMU | DEP |
| ARM | ARM 1 to ARM 7 | ✗ | ✗ | ✗ |
| | ARM 7EJ | ✗ | ✗ | ✗ |
| | ARM Cortex R | ✓ | ✗ | ✓ |
| | ARM Cortex M | ~ | ✗ | ✓ |
| PIC | PIC 10 to PIC 24 | ✗ | ✗ | ✗ |
| | dsPIC | ✗ | ✗ | ✗ |
| AVR | ATiny | ✗ | ✗ | ✗ |
| | ATmega | ✗ | ✗ | ✗ |
| | ATxmega | ✗ | ✗ | ✗ |
| 8051 | Intel MCS-51 | ✗ | ✗ | ✗ |
| | Infineon XC88X-I | ✗ | ✗ | ✗ |
| | Infineon XC88X-A | ✗ | ✗ | ✗ |
| MSP430 | MSP430x1xx to MSP430x6xx | ✗ | ✗ | ✗ |
| | MSP430FRxx | ✓ | ✗ | ✗ |

✓: *it is supported by all microcontrollers in the given family.*
~: *it is supported by some microcontrollers in the given family.*
✗: *it is not supported by any of them.*

automatically finding vulnerabilities on a large scale. The main idea of fuzzing is executing the software in a test environment with random inputs to look for vulnerability-exposing behaviors such as crashing or hanging. Indeed, such behaviors are immediate consequences of faulty states, and the ability to observe them is the prerequisite for fuzzing to work. In general purpose computers, equipped with OS security mechanisms and hardware features such as stack canaries, Address Space Layout Randomization (ASLR) and Memory Management Unit (MMU), memory violations trigger a crash upon a fault. There are three strategies to observe such crashes: (1) Observing exit status: the execution of the device or application under test is terminated and an error message is generated for tracing. (2) Catching the crashing exception: the crashing signal can be caught by overwriting an exception handler. (3) Leveraging mechanisms provided by the OS: the OS-level debugging interfaces such as ptrace can be used in order to observe application execution and detect crashes.

Fuzzing methods could be categorized depending on how much information is collected and used from the application under test to generate the input as: (1) Blackbox fuzzers, such as Boofuzz [7], which have no information about the target application and blindly test a large number of random inputs. However, this approach is not very effective for uncovering bugs in deep parts of the code. (2) Whitebox fuzzers, which use expensive program analysis techniques such as dynamic taint analysis and symbolic execution for collecting and utilizing feedback from application execution in order to guide the generation of the inputs. (3) Greybox fuzzers, which are the most scalable and practical fuzzers [8], [9], [10], provide a middle ground between blackbox and whitebox fuzzers by inferring limited information about the application extracted with lightweight analysis techniques like code coverage and feeding that information back to guide the input generation process.

The best known Greybox fuzzer is the American Fuzzy Lop (AFL) [8], which leverages execution tracing information to tailor input generation. Since collecting and analyzing full application traces incurs high overhead, AFL applies a more practical strategy by tracking edge-coverage as an approximation of an application execution trace.

## 2.2 Bare-Metal Embedded Devices

Among different types of embedded devices that are widely used in cyber-physical systems, bare-metal devices are designed for low cost and low power operations. Such devices are deployed in many application areas ranging from smarthome to automotive, from industrial control systems to medical devices. In bare-metal devices, applications directly run on the hardware without having any underlying abstraction such as an OS. In fact, each executable firmware of the bare-metal devices is a statically linked binary image that provides both the low level services and the application logic. However, given the nature of bare-metal devices, traditional security mechanisms from general purpose computers are not readily applicable in such devices. This becomes clear when one considers that bare-metal devices have tight constraints on runtime, energy usage, and memory usage. For instance, this class of devices rarely provide an MMU; thus any code has access to all memory and peripherals without any protection. Consequently, compromising one firmware module enables an adversary to arbitrarily redirect the control-flow of firmware or directly overwrite sensitive data with no observable sideeffects.

As a more concrete investigation of the hardware security feature support (i.e., MMU, MPU, and DEP), we conducted an analysis of 29 SoC core families. Our selection aims to provide a representative sample of major architectures and vendors in the embedded space across industry verticals including unmanned aerial vehicle (UAV), unmanned ground vehicle (UGV), remotely operated underwater vehicle (ROV), real-time 3D printer controllers and real-time Internet of Things (IoT) devices.

According to our analysis, none of the SoCs is designed to employ MMU. A number of SoCs optionally provide basic memory protections using Memory Protection Unit (MPU). However, even with the existence of MPU, configuring it from the application is not a straightforward task, leading the developers to ignore using this functionality. Table 1 summarizes the results of our analysis by mapping out core families architectural style and hardware security functionalities.

### 2.3 Challenges of Fuzzing Bare-Metal Devices

Binary image of a bare-metal firmware is often tailored to the microcontrollers with limited computing and memory resources. In fact, due to its closed source and architectural diversity, it becomes extremely challenging to conduct fuzzing on such firmware. In what follows, we summarize four main challenges in fuzzing bare-metal firmware.

*Fault Detection.* Fault detection in bare-metal devices remains a challenge [3]. General purpose computers gain a significant amount of fault detection capability from the visibility provided by plenty of deployed mechanisms such as segmentation faults caused by an MMU, making many memory corruptions much less silent. Most bare-metal devices, instead, do not have such mechanisms owing to their limited I/O capabilities, constrained cost, and limited computing power. In fact, most memory corruptions events are silent and do not lead to any observable and immediate crash of the firmware. Consequently, the firmware continues the execution with no visible effect or the fault might only become noticeable at a later point (i.e., I/O error), which is very difficult to debug. It is challenging to infer if the crash was because of an early memory violation or an I/O error.

*Memory Corruption Identification.* Identifying the root cause of a crash and type of potential memory corruption vulnerability is a time-consuming and challenging effort, causing a disproportion between discovering a crash and patching the underlying application vulnerability. State-of-the-art IoT fuzzers only focus on producing crashes as much as possible with no information about the actual origin of the crash, sometimes overwhelming security analysts to patch them. In addition, this situation can be worsened when one unique vulnerability results in several crashing inputs: a fuzzer can discover multiple execution paths to a crash, whereas the vulnerability is always the same. Therefore, it would be required to investigate a large number of potential vulnerabilities.

*Memory Sanitization.* Sanitization techniques [11], [12], [13], [14], [15] can be combined with fuzzing methods to improve fault detection capability of fuzzer and discover memory corruptions as they happen. Furthermore, sanitization techniques provide more detailed information on the location and root cause of the crash. These techniques instrument applications with the aim of enforcing memory safety policies. Sanitization techniques detect dereferences of pointers that either do not access their intended referent (i.e., spatial memory safety violations), or that access a referent that is no longer valid (i.e., temporal memory safety violations). Specifically, sanitization techniques insert inlined reference monitors (IRMs) into the application and monitor every memory accesses and memory object (de)allocations instructions. This can be done either at source code or binary level. However, owing to the fact that bare-metal firmware are often proprietary and their source code is not available, binary sanitization is the only viable solution.

Indeed, binary images of bare-metal firmware are often available to the analyst since they can be acquired by downloading and unpacking update packages available on many vendors' websites or directly extracting from the physical device using debugging port (e.g., JTAG interface). Dynamic binary sanitization tools [12], [15], [16] transform binaries as they are executing. In addition to the significant run-time and space overhead, dynamic binary sanitizers do not produce a standalone binary and the output instrumented code is tailored to the tool's runtime environment. In fact, the instrumented binary cannot be used for subsequent executions and the sanitization process has to be done again each time the application executes. These problems can essentially be attributed to the dynamic translation process and they can be addressed by instrumenting applications statically using a static binary sanitization tool, which we believe is a promising solution for our requirements. Unfortunately, at the time of writing none of the proposed binary sanitization tools provides support for bare-metal firmware.

*Benchmark Suite.* Defining an experimental setup for evaluating the effectiveness of a fuzzer is challenging. Specifically, because of the randomness and domain specialization in fuzzing process, evaluating and comparing fuzzers by just running them on a set of firmware, without any other constraint, can produce misleading results. We point out that fuzzing mechanisms must consider the following observations in order to have adequate evaluations:

(1) All modern fuzzing mechanisms rely on randomness for input generation procedure. Thus, in every single run, the fuzzer may discover different number of crashes than the last since random choices make the fuzzer to explore different paths. As a result and as mentioned by Klees *et al.* [5], it is inadequate if we simply run two fuzzers, A and B, over the same firmware and compare their performance. Rather, it is required to run both A and B for a long period of time so that fuzzers are able to cover all the choices possible. Of course, this is not a solid solution as the mutation space of an input is infinite, therefore there are always inputs that are left out from the experiment. To reduce the chances of an incorrect result, the evaluation must include sufficiently many trials such that a statistical test is performed to validate the claims.

(2) Fuzzing effectiveness may vary with the firmware under test, therefore it is essential to conduct the evaluation on a diverse, representative benchmark suite. However, the availability of bare-metal firmware is very limited. Typically, the ones used for evaluating IoT fuzzers are compiled from code samples provided by board manufacturers or extracted from real devices. The former firmware are publicly available but demonstrate limited functionalities compared to the realistic firmware deployed in market devices. The latter ones, instead, present the issue that they cannot be publicly distributed, making the experiments not repeatable.

(3) Most IoT fuzzers only rely on the number of crashing inputs as an evaluation metric for fuzzing performance. However, there is not a one-to-one correspondence between crashing inputs and actual memory vulnerabilities. More precisely, the randomness in input mutation may generate different inputs that cause crashes by triggering the same memory vulnerability. There are some proposed heuristics for de-duplicating crashes such as stack hashes [4] and coverage profiles [20], [21]. As pointed out by Klees *et al.* [5], such heuristics are insufficient to identify unique vulnerabilities.

TABLE 2
A Summary of Conducted Experimental Evaluations by State-of-the-Art IoT Fuzzers

| Observations | IoTFuzzer [17] | Firm-AFL [18] | FirmFuzz [19] | WCYCWYC[3] | $P^2$IM [10] | HAL-Fuzz [9] |
|---|---|---|---|---|---|---|
| Firmware in Benchmark | 17 Real | 51 Real | 32 Real | 1 Artificial | 10 Real | 13 Artificial & 5 Real |
| Evaluation Metric | #CI & #UV | #CI & #UV | #UV | #UV | #UV | #CI & #UV |
| Fuzzing Timeout | 24H | 24H | 16M:42S | 1H | 24H | 10H-23D:14H |
| Multiple Trials | NR | 10 | NR | 100 | NR | NR |

*Firmware in Benchmark row represents the number of realistic (Real) and artifictial firmware in the used benchmark. Evaluation Metric represents considered metrics, number of crashing inputs (#CI) or unique, discovered vulnerabilities (#UV), for evaluating fuzzing performance. Fuzzing Timeout represents the reported time of each fuzzing session in seconds (S), minutes (M), hours (H), and days (D). Multiple Trials represents number of trials in evaluation. NR means that this item is not reported in the paper evaluation.*

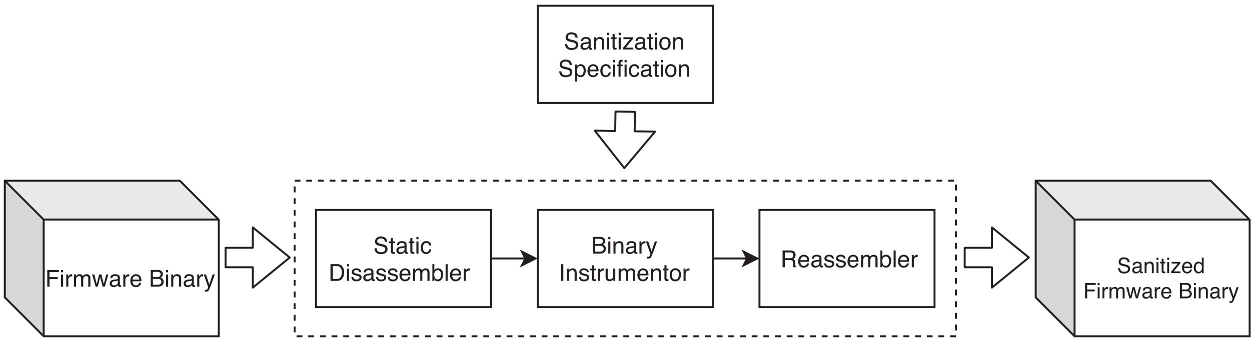


Fig. 1. Pipeline of our proposed firmware sanitizer.

Alternatively, some researchers manually investigate crashes to discover unique vulnerabilities in firmware (e.g, reporting CVEs). However, this evaluation approach requires a security analyst with extensive domain expertise. In addition, suppose that a firmware presents n vulnerabilities of type x and m of type y, with n > m. If n and m are not known, a fuzzer A, which is specialized in detecting only vulnerabilities of type x, may detect n crashes in t seconds, while a general fuzzer B detects in total less than n crashes in the same amount of time. In spite of the numbers showing A performs better than B, in reality, the analyst using A would not be able to detect type y vulnerabilities while the analyst using B detects them if B is executed more than t seconds. As a result, having a benchmark suite with ground truth allows us to correctly measure how many vulnerabilities have been missed, giving better insights about the fuzzing performance.

According to our study, state-of-the-art IoT fuzzers conducted experimental evaluations inadequately that induce wrong or misleading conclusions. For instance, they carried out the evaluation based on the number of crashing inputs or unique discovered vulnerabilities without using a benchmark suite with ground truth. Table 2 summarizes the results of our study.

## 3 Instrumentation for Vulnerability Discovery and Identification

Fig. 1 illustrates a high-level overview of our approach, with the different components and their interactions. There are three main components: the static disassembler, the binary instrumentor, and the reassembler.

The first component of our framework pipeline, the static disassembler, parses the executable region of the binary file from beginning to the end and decodes all encountered bytes into their raw textual representation. Two popular approaches [22], [23], [24] for disassembling binaries are linear sweep and recursive descent. Linear sweep goes through the entire executable section and decodes all encountered bytes as instructions, while recursive descent disassembles all reachable code in the binary by following control flow transfers (e.g., jumps and calls). As shown by Andriesse *et al.* [25], the linear sweep approach outperforms tools that use more sophisticated methods. Therefore, we applied a linear sweep disassembly algorithm to our evaluation set.

The second component is binary instrumentor that instruments the firmware binary statically based on sanitization specifications. Sanitization specification determines what instructions will be inserted or replaced in order to enforce memory safety policies by embedding inlined reference monitors (IRM). In other words, our framework statically instruments every memory access with a runtime check to verify if it is an access to an allowed address. If not, our fault handler raises a crash close to the location of the vulnerability in order to trigger the fuzzer. Moreover, sanitization specification provides the analyst with context information on crashing causes and type of potential memory corruption vulnerability. The third component, the reassembler, takes the instrumented assembly code and reassembles it as a working binary using off-the-shelf assemblers.

The proof-of-concept implementation of our framework provides support for the ARMv7-M architecture, which covers the widely deployed Cortex-M(3, 4, and 7) microcontrollers in embedded platforms [26], [27]. In the following sections, we describe the most important and challenging aspects of the proposed framework design and implementation.

### 3.1 Binary Instrumentation

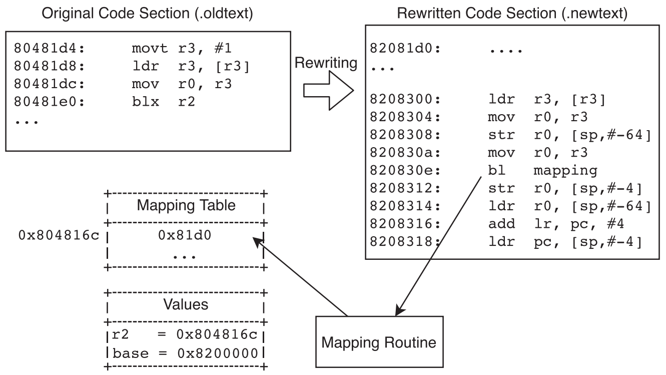
The proposed binary instrumentor component takes as input the disassembled firmware and inserts or replaces instructions based on the sanitization specification. However, due to the lack of linkage information, instrumenting the input disassembled firmware is not as straightforward as editing source code or compiler-generated assembly file. More precisely, the assembly file that is produced by compilers maintains symbol and relocation information to ensure that application elements can correctly refer to each other. Therefore, when instructions are inserted or replaced, the compiler will rearrange code and data in memory and manage references between them. However, since symbol and relocation information are discarded and addresses are hard-coded after the linking process, instrumenting disassembler-generated assembly causes addresses to change and breaks the firmware image file.

There are three main challenges in relocation procedure to avoid breaking the binary file. The first challenge is recognizing static addresses. There is no syntactic distinction to disambiguate between reference and scalar type for immediate values and updating references to the new targeted addresses. The second challenge is relocating static addresses after instrumentation. Indeed, insertion of instructions into, or removal of instructions from disassembly code can break these static addresses. The third challenge is determining dynamically referenced memory addresses. Contrary to static memory addresses that are explicit, the target addresses of some references are computed dynamically at runtime and they can not be updated statically.

The crux of instrumenting binaries is the ability to relocate any binary code without any relocation and meta-data information. Our approach duplicates the code section, with the old copy (called .oldtext) as a read-only section and the new copy (called .newtext) as an executable code section containing rewritten instructions. It then adjusts all the target addresses of data/code pointers to ensure that they point to their targeted locations. Insertion of instructions may push a referenced instruction/data beyond the reach of the instruction referencing it. As a remedy, the instrumentor component expands all referencing instructions with a short encoding by their substitutes with longer encoding which allows for larger offsets.

Due to the fact that there is no need to perform instrumentation on the original data space, the instrumentor can preserve .oldtext and data sections at their original addresses intact. By doing so, we may easily ignore and handle data pointers in the rewritten code section (i.e., .newtext) and they continue to behave correctly. Regarding code pointers, the instrumentor relocates target addresses of all branch instructions to the new addresses while rewriting them in the .newtext section. Specifically, direct branch instructions can be statically rewritten by changing their offset. However, indirect branch instructions have multiple possible target addresses and therefore needs some sort of target-prediction mechanism. In contrast to the related work [28], [29], [30], we believe that although it is challenging to statically determine exact target addresses of indirect branches, we can instead apply an efficient dynamic lookup at runtime. Indeed, the exact targets of control flow transitions are known at runtime.

Our instrumentor component adds a level of indirection by redirecting the target addresses of indirect branch instructions to the rewritten new addresses through a mapping routine at runtime. Essentially, the mapping table is created during the rewriting process, and it contains each possible target address in the .oldtext mapped to the corresponding address in the .newtext section. For example, as illustrated in Fig. 2, the instrumentor rewrites original instructions from the .oldtext section along new inserted instructions in the .newtext section with new base address 0x8200000. It replaces every indirect branch instruction (*blx r2*) with the mov and direct call instructions. Precisely, the mov instruction stores the runtime value of indirect branch target address (0x804816c) into the register r0 and the direct call (*bl mapping*) goes to the mapping routine in order to search for the offset corresponding to the old target address in the mapping table (0x81d0). At the end, mapping routine returns new translated target address (0x8200000 + 0x81d0 = 0x82081d0) for jumping (*ldr pc, [sp, #-4]*) to it accordingly.


Original Code Section (.oldtext)
Rewriting
Rewritten Code Section (.newtext)
Mapping Table
0x804816c
0x81d0
...
Values
Mapping Routine

```
80481d4:     movt r3, #1
80481d8:     ldr  r3, [r3]
80481dc:     mov  r0, r3
80481e0:     blx  r2
...
```

```
82081d0:     ....
...

8208300:     ldr  r3, [r3]
8208304:     mov  r0, r3
8208308:     str  r0, [sp,#-64]
820830a:     mov  r0, r3
820830e:     bl   mapping
8208312:     str  r0, [sp,#-4]
8208314:     ldr  r0, [sp,#-64]
8208316:     add  lr, pc, #4
8208318:     ldr  pc, [sp,#-4]
```

```
r2   = 0x804816c
base = 0x8200000
```

Fig. 2. The binary instrumentor component redirects all indirect branch instructions (e.g., blx r2) to the mapping routine which looks for new offset (0x81d0) corresponding to the old target address (0x804816c) in the mapping table.

*Implementation Details.* We implemented the binary instrumentor component on top of Capstone disassembler framework [31], spanning 1710 SLOC in Python language. Our instrumentor utilized pyelftools [32] open source framework to parse the ELF data structures. It leverages LIEF framework [33] in order to edit the header of firmware ELF file and create a new code segment containing the .newtext section and mapping routine. It also used pwntools [34], an open-source binary analysis framework, as the platform for reassembling the instructions.

### 3.2 Sanitization

Our framework proposes a static binary sanitization technique by leveraging its binary instrumentor component. Inspired by the most widely adopted sanitization techniques, i.e., AddressSanitizer [13] and Valgrind's Memcheck [12], the framework uses a metadata store that maintains the status for each byte of the addressable memory; it inserts so-called red-zones between memory object representing out-of-bounds memory and marks it as invalid memory in the metadata store. The framework instruments every memory instruction in order to consult this metadata store whenever the firmware attempts to access memory, identifying if the memory access is valid or not. Any access to a red-zone or to an unallocated memory region is considered as a potential memory corruption vulnerability and raises a crash close to the location of the vulnerability for triggering the fuzzer.

More precisely, our framework surrounds every function's stack frame with a redzone, enabling it to detect overflows on the stack. Thus, it adds new bytes for the redzone in the metadata store to disallow access to it. Then, due to the fact that the stack frame is implicitly freed upon function return and may be reused by the next function call, we identify each function exit and unpoison the redzone in the metadata store prior to exiting the function. Furthermore, our framework keeps track of all heap blocks issued and allocates memory for the redzones in the metadata store. It then marks that allocated memory as inaccessible heap left and right redzones. Thus, if the firmware attempts to read or write from those memory addresses, all such attempts would be detected because they will all land on the left and right redzones (i.e., heap underflow and heap overflow respectively). Finally, when we deallocate each heap block, our framework frees the corresponding part in the metadata store.

Note that we may miss vulnerabilities when the overflow happens inside the function's stack frame. Unlike source-based sanitizers, since the firmware binaries are stripped, our approach does not have access to variable scope and type information. This limitation is common to all binary-only approaches. However, our approach granularity on the heap is equivalent to source-based sanitizers, enabling it to detect overrunning and underrunning heap blocks.

Additionally, the sanitization records all memory accesses for crash triaging. When a crash is discovered, it provides the analyst a chain of executed memory instructions in chronological order along with the type of potential memory corruption vulnerability. Indeed, according to the context information on the erroneous behaviors, our proposed framework identifies different types of spatial and temporal memory corruption vulnerabilities including: (1) Overrunning the top of the stack. (2) Overrunning and underrunning heap blocks. (3) Accessing memory after it has been freed. (4) Using memory values that have not been initialized or that have been derived from other uninitialized values. (5) Incorrect freeing of heap memory, such as double-freeing heap blocks. For example, our framework keeps track of the blocks allocated and deallocated by the firmware by calling *malloc* and *new* functions. Since these functions have specific binary pattern in memory, their locations and instructions calling them can be determined by our framework. Indeed, our framework records memory addresses passed to these functions as arguments. Assume a firmware has freed a block in a specific memory address twice. The sanitization process makes the firmware to crash if the vulnerability is triggered. Afterwards, it identifies and reports a double-free vulnerability by analyzing recorded memory (de)allocations.

*Implementation Details.* We implemented our binary sanitization technique on top of the binary instrumentor component with 825 SLOC in the Python language. We use the disassembly extracted from the disassembler component to store all memory accesses and object (de)allocations instructions in the sanitization specification file. Then, the binary instrumentor interprets the sanitization specification file in order to instrument all these instructions with memory check instructions. The memory check instructions consult the metadata store for validating the intended memory access. More precisely, memory check instruction extracts the target address of intended memory access and calculates the address of the corresponding metadata byte to check if it is an access to an allowed address, i.e., not a redzone. Due to the fact that checking allocation status for every single byte of memory is significantly expensive, our approach applies an efficient metadata management mechanism like AddressSanitizer. In fact, it maps eight bytes of memory to a single byte of metadata. By doing so, the metadata mapping formula can be represented with (1) where meta_base is the base address of the metadata and block_addr is the address of the memory block.

$$meta\_addr = meta\_base + (block\_addr >> 3). \tag{1}$$

## 4 Ground-Truth Benchmark Suite

In this section, we introduce a ground-truth benchmark suite that provides a valid and reliable metric to evaluate and compare IoT fuzzers. Our benchmark framework allows to inject the type of vulnerability for which we want to evaluate the IoT fuzzer in the position of the program we choose. Thus, the test suite can assess IoT fuzzers' capabilities in detecting different types of vulnerabilities. Furthermore, for each vulnerability, the benchmark can provide the correct input which triggers it.

Our benchmark framework addresses all the challenges highlighted in Section 2.3. In particular, it takes a set of firmware and injects the chosen vulnerabilities in each of them. Then, it provides an input that allows the execution of such vulnerable code. Indeed, for evaluating IoT fuzzers, it is required to have a firmware that has at least a communication channel with the external world to receive inputs. The framework can be applied to both open source and binary firmware. In the former case, the vulnerability injection process is automatic and no human effort is required, while, for the latter, the injection point should be manually indicated.

An overview of the proposed benchmark framework is shown in Fig. 3. It consists of several steps that bring to the creation of a vulnerable firmware and a set of inputs that triggers each injected vulnerability. The pipeline is explained below.

*Firmware Selection.* We selected firmware for the benchmark based on the following criteria: (1) The firmware in the benchmark must be realistic, full-fledged, and deployed in market devices. (2) The firmware in the benchmark must be diversified in functionality. (3) The benchmark must cover the use of various peripherals such as LCD display, microphone, camera, serial port, Ethernet and SD card to represent realistic interactions of IoT devices.

The current version of our benchmark suite contains 18 representative firmware, which we summarize in Table 3. All of these firmware images are provided with the development boards and written by STMicroelectronics [35] except Drone, CNC, Gateway, Robot, and Reflow Oven firmware which are collected from [36], [37], [38], [39], [40] respectively by P$^2$IM's authors [10].[2]

2. Regarding the license of the collected firmware, the ones developed by STMicroelectronics can be redistributed as a benchmark. However, they must be used and executed solely and exclusively on or in combination with a microcontroller or a microprocessor device manufactured by or for STMicroelectronics. Furthermore, the Gateway, Drone, CNC, Robot, and Reflow Oven firmware can be modified, redistributed and used in any types of devices.

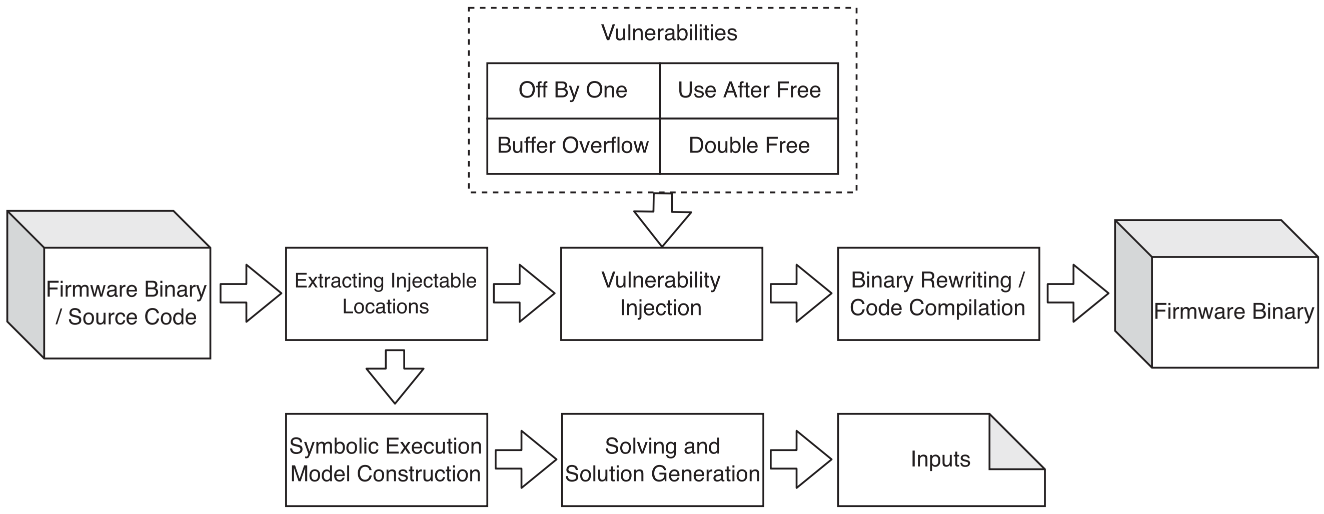

Vulnerabilities
Off By One
Use After Free
Buffer Overflow
Double Free
Firmware Binary / Source Code
Extracting Injectable Locations
Vulnerability Injection
Binary Rewriting / Code Compilation
Firmware Binary
Symbolic Execution Model Construction
Solving and Solution Generation
Inputs

Fig. 3. Benchmark generation pipeline.

TABLE 3
Bare-Metal Firmware, Their Targeted MCUs, and Used Peripherals Incorporated Into Our Benchmark Framework

| Firmware | MCU | Peripherals | Firmware | MCU | Peripherals |
|---|---|---|---|---|---|
| Audio-Playback | STMF479I-Eval | Clock, GPIO, USB, I2C | LibJPEG_Encoding | STMF4Discovery | Clock, GPIO, SD-CARD, DSI |
| PLC | STMF429ZI | Clock, Timer, WiFI, UART, SPI | LibJPEG_Decoding | STMF4Discovery | Clock, GPIO, SD-CARD, DSI |
| Drone | STMF103RB | Clock, GPIO, I2C | CNC | STM32F429ZI | Clock, GPIO, UART, USB, I2C |
| TCP_echo_Client | STMF479I-Eval | Ethernet, Clock, GPIO, EXTI | Gateway | STM32F103RB | GPIO, UART, I2C |
| TCP_echo_Server | STMF479I-Eval | Ethernet, Clock | Robot | STM32F103RB | Clock, GPIO, UART, I2C |
| UDP_echo_Client | STMF479I-Eval | Ethernet, Clock, GPIO, EXTI | Camera-USB | STMF479I-Eval | Clock, GPIO, USB, DSI |
| UDP_echo_Server | STMF479I-Eval | Ethernet, Clock | Reflow Oven | STMF103RB | GPIO, UART, I2C |
| FatFs_uSD | STMF4Discovery | Clock, GPIO, SD-CARD | mbed-TLS | STMF401RE | Ethernet, Clock, GPIO, EXTI, DSI |
| LCD_Display | STMF479I-Eval | Clock, GPIO, SD-CARD, DSI | LCD_Animate | STMF479I-Eval | Clock, GPIO, SD-CARD, DSI |

Audio-Playback is a firmware that reads and plays audio files from USB. PLC (Programmable Logic Controller) is a firmware for controlling critical processes in an industrial environment. LCD_Display is a firmware for reading bitmap pictures from an SD card and displaying them on the LCD display. LCD_Animate creates the effect of animation by displaying multiple layers of bitmap images. Drone firmware is used as an autopilot controller in a quad-copter. TCP/UDP-Echo-Client/Server firmware implement four TCP/UDP echo client/server over Ethernet. FatFs-uSD firmware implements a FAT file system on an SD card. LibJPEG_Decoding is a firmware for reading a jpeg file from SD card memory, decoding it, and displaying the final BMP image on the LCD, while LibJPEG_Encoding is developed for reading a BMP file from SD card, encoding it, and saving the jpeg file in SD card. CNC firmware implements stepper-motor control routines in several laser cutters, and 3D printers. Gateway firmware implements a communication protocol in a gateway device. Robot is the motion controlling firmware in a robot. Camera_USB firmware leverages the camera module to display pictures in a continuous mode on LCD and save them in a USB device. Reflow oven is used in a reflow oven controller that assembles printed circuit boards (PCB). mbed-TLS is an SSL client firmware that implements mbedTLS crypto library and LwIP TCP/IP stack on IoT devices.

*Extracting Injectable Locations.* Our benchmark framework first finds a list of all possible locations in the firmware source code to inject vulnerabilities. These injection points require to be accessed and triggered by specific inputs. Therefore, the framework extracts all the functions that receive inputs from external sources such as network, UART, GPIO, and other peripherals in general (i.e., input functions). Since such functions deal with inputs, they are good targets to inject vulnerabilities. We identify these targets using the Hardware Abstraction Layers (HAL). Mainly due to the fact that HALs are usually provided by chip vendors and various third parties in order to simplify firmware developers' jobs, such abstractions can be leveraged for identifying input functions. Specifically, we match a set of input functions from HALs' documents with those implemented in the firmware. Every match is a candidate for the injection point.

*Vulnerability Injection.* Our benchmark framework allows the injection of different types of vulnerabilities by inserting snippets of self-triggering vulnerable codes. To evaluate IoT fuzzer performance, the benchmark framework injects the vulnerability in a way that only a particular input can trigger it. In other words, the vulnerable code is inserted in a selection statement where the condition is specifically crafted to avoid accidental executions requiring a non-trivial input. Also, this condition can potentially be customized to assess specific fuzzer capabilities such as the capability of handling search-hampering features in the code (e.g., magic values and checksums) or the capability of reaching a desired depth level by concatenating several conjunctive conditions.

*Code Compilation.* The proposed framework compiles firmware source code after finishing the vulnerability injection process in order to obtain the vulnerable firmware binary.

*Symbolic Execution Model Construction.* After injecting the vulnerability, the benchmark framework constructs a symbolic execution model that allows the inference of a valid, triggering input. In particular, the outcome of this step is a logic formula that considers all the conditions and loops which the variable, containing the input, undergoes before reaching the vulnerable code. As mentioned earlier, the framework supports the injection of multiple vulnerabilities in the same firmware. This requires a specific model for each injection so that a specific input can be constructed.

*Solving and Solution Generation.* In the last step, our framework checks the satisfiability (SAT) of the symbolic execution model. SAT is a property of the logic formulas which is confirmed when an interpretation, that makes the formula *true*, exists. If the model is SAT, then the reachability of the vulnerable code is formally verified. Afterward, in a successful case, the framework generates a solution that represents a valid input that leads firmware execution to the vulnerable code. This solution is the instance of an interpretation that makes the formula *true*.

*Implementation Details.* The benchmark framework implementation consists of 300 SLOC of Python code. In addition, the injected vulnerabilities are written in 106 SLOC of C code. The framework uses two solutions, depending on the firmware source code's availability, to find the injection point and generate the corresponding input for accessing that. In the following, each step of our implementation is described.

First, the framework extracts all the input functions from HALs' documentations and match them with the ones implemented in the firmware code. This strategy works well for most but not all the scenarios. In particular, some HALs interact with some certain input peripherals as event based devices, and they implement the input reading mechanisms by means of callback functions. In fact, input functions are user-defined as callbacks and get called once an input is available. Therefore, the framework cannot obtain such functions directly from standard HALs' documentations. Nevertheless, each callback function must be registered to make the HAL aware of it, and this typically happens by passing the callback function as an argument to a specific function. Our benchmark framework leverages this observation to find callback registration functions and extract callback function as an input function for vulnerability injection.

After finding the injection point, our framework injects one of the supported memory corruption vulnerabilities, i.e., stack-based buffer overflow, stack off-by-one, double free, and use-after-free. It is worth noting that the current version of our framework supports these vulnerabilities since they represent many of the exploited IoT firmware in the wild and, in some instances [2], allow the complete takeover of the device. Finally, the framework adds an `assert (FALSE)` statement before the vulnerability location and runs the CBMC [41] model checker. By doing so, the model checker will stop at the assertion and provide the inputs which have brought the execution here.

Alternatively, if the firmware is available only in its binary form, then the process of finding a suitable injection point requires manual intervention. In particular, we manually have to identify the function that reads the input. Also, in this case, for generating the input that triggers the injected vulnerability, we use the *angr* symbolic execution engine [42], [43] to construct the symbolic execution model and *Z3* [44] as SAT solver.

## 5 Evaluation

This section presents the evaluation of various aspects of our proposed approach. First, we evaluate the correctness of the instrumentation method against our ground-truth benchmark in Section 5.1. Second, we measured the runtime and space overhead of our instrumented firmware binaries in Section 5.2. Finally, we demonstrate the effectiveness of sanitization method for improving fault detection capability and identification of memory corruption vulnerabilities in existing IoT fuzzers in Section 5.3.

### 5.1 Feasibility and Correctness

We performed an evaluation on the correctness of the proposed instrumentation method by comparing the output of the original and instrumented firmware binaries. Particularly, we applied the instrumentation method on each benchmark firmware and obtained the instrumented version of them. Afterwards, we executed the instrumented firmware by the test suite shipped with the original firmware in order to show that all instrumented firmware execute correctly and produce identical output to the original one. Using this data, we can be confident of the correctness of the implementation of our approach design. Note that, we did not attempt to exhaustively run all the execution paths of the two firmware versions and simply used the same configuration to run them.

Table 4 also lists the modifications made by instrumentor component to our benchmark firmware. Column *Dir. Inst.* reports the number of direct calls and jumps that are modified by adjusting their target address statically. Beside, column *Ind. Inst.* reports the number of indirect calls and jumps that are modified by redirecting them to the mapping routine for obtaining new target addresses dynamically. Furthermore, data shows that our approach has an acceptable impact on binary size when delivering instrumentation. Column *Size Inc.* reports the incurred size expansion on the code sections of instrumented firmware binaries. Note that ARM instruction set has a fixed instruction length and all the instructions are either two or four bytes. This overhead is positively correlated with the number of indirect calls and jumps since we instrument them with mapping instructions. We do not include the mapping table in *.newtext* column in the table owing to the fact that it is always four times larger than the *.oldtext* column: mapping table keeps 4-byte data for every single byte in .oldtext.

### 5.2 Performance

To measure the performance impact of our framework sanitization, we compare the runtime between an uninstrumented benchmark version and a sanitized version. We start profiling all firmware just before the main function begins execution and stops at a hard-coded point. Fifteen runs of each firmware were averaged and in all runs the standard deviation was less than 2%. As an instance, we run FatFS-uSD uninstrumented and sanitized firmware for

TABLE 4
Statistics of Firmware Binaries Instrumented by Our Approach

| Firmware | Dir. Inst. | Ind. Inst. | Mem. Read | Mem. Write | .oldtext (KB) | .newtext (KB) | Size Inc. (%) |
|---|---|---|---|---|---|---|---|
| Audio-Playback | 5290 | 259 | 14529 | 5777 | 110 | 533 | 384 |
| LCD_Display | 2107 | 103 | 2097 | 1440 | 30 | 141 | 370 |
| LCD_Animate | 2097 | 103 | 2100 | 1436 | 30 | 140 | 366 |
| FatFs_uSD | 1654 | 79 | 1379 | 978 | 21 | 98 | 366 |
| TCP_Echo_Client | 3728 | 132 | 4341 | 2219 | 52 | 223 | 328 |
| TCP_Echo_Server | 3566 | 132 | 7132 | 6762 | 51 | 218 | 327 |
| UDP_Echo_Client | 3471 | 132 | 4142 | 2144 | 49 | 215 | 338 |
| UDP_Echo_Server | 3381 | 130 | 4036 | 2075 | 48 | 208 | 333 |
| Camera-USB | 2906 | 161 | 2725 | 1270 | 41 | 187 | 356 |
| mbed-TLS | 8256 | 341 | 8904 | 4293 | 116 | 462 | 267 |
| PLC | 1451 | 190 | 1850 | 1057 | 22 | 110 | 400 |
| LibJPEG_Encoding | 4586 | 639 | 6867 | 3762 | 81 | 367 | 353 |
| LibJPEG_Decoding | 3580 | 447 | 4995 | 2850 | 61 | 277 | 354 |
| CNC | 2845 | 174 | 3882 | 1542 | 45 | 163 | 262 |
| Gateway | 3253 | 315 | 3412 | 1723 | 40 | 183 | 357 |
| Robot | 2457 | 136 | 3610 | 1770 | 39 | 164 | 320 |
| Reflow Oven | 1912 | 206 | 2299 | 1270 | 28 | 135 | 382 |
| Drone | 2052 | 113 | 1595 | 900 | 28 | 105 | 275 |

formatting the SD card, creating a file, writing 2,048 bytes to the file, and verifying the contents of the file fifteen times. Fig. 4 illustrates the performance results. The average runtime overhead for the sanitized firmware is 53.7%. Our approach imposes more runtime overhead on Audi-Playback, TCP-Echo-Server, and mbed-TLS benchmarks compared to the other cases. This is due to the fact that they are memory-intensive firmware resulting in a large number of memory check instructions. Although these runtime slowdowns are not negligible, we believe that they are acceptable since they are only incurred at the testing time not the actual deployed firmware on the field.

We also measure how long it takes our framework to sanitize benchmark firmware. Fig. 5 presents the processing time for our benchmark firmware binaries. Indeed, large firmware images such as mbed-TLS take more time to process. The median processing time for the sanitization phase is 334 seconds. We interpret this as an encouraging result which makes our framework a tool totally practical for sanitizing bare-metal firmware in the large-scale.

## 5.3 Effectiveness

This experiment evaluates the effectiveness of proposed sanitization in discovering and identifying memory corruption vulnerabilities. To understand how it improves fault detection and crash triaging capabilities in the state-of-the-art IoT fuzzing methods, we executed HAL-Fuzz [9], [45] and P$^2$IM [10] fuzzers over sanitized and original versions of benchmark firmware. HAL-Fuzz and P$^2$IM are the state-of-the-art IoT greybox fuzzers designed and developed atop AFL-Unicorn and AFL fuzzers respectively. For each firmware, we injected four vulnerabilities (i.e., one Stack-based buffer overflow, one Off-by-one, one Double Free, and one Use-after-free vulnerability) using our benchmark framework and conducted 10 identical fuzzing sessions lasting 24h each. In order to have a fair evaluation, fuzzing parameters were identical across all the sessions. Indeed, each fuzzing session was configured with the same time-out and memory limit and bootstrapped with the same set of seeds.

The fuzzer effectiveness is measured in terms of vulnerabilities reached, triggered, and detected. Our framework provides some information when the sanitized code is exercised, allowing us to determine whether a vulnerability is reached. When the dataflow of the fuzzer-generated input satisfies the vulnerability's trigger conditions, the vulnerability is triggered. When a vulnerability is triggered, the fuzzer should report it as a fault or crash, allowing us to evaluate the fuzzer vulnerability detection capability.

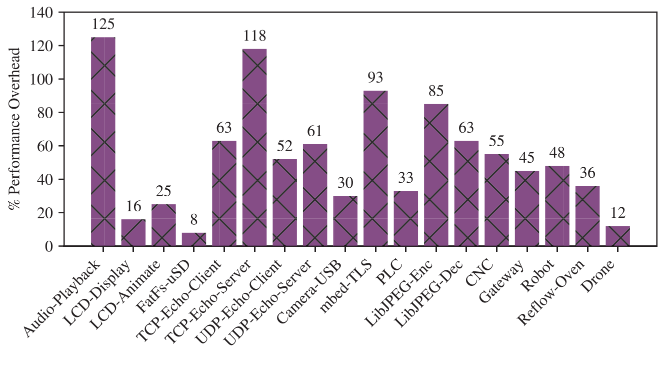


Fig. 4. Performance impact of proposed sanitization approach over the benchmark suite.

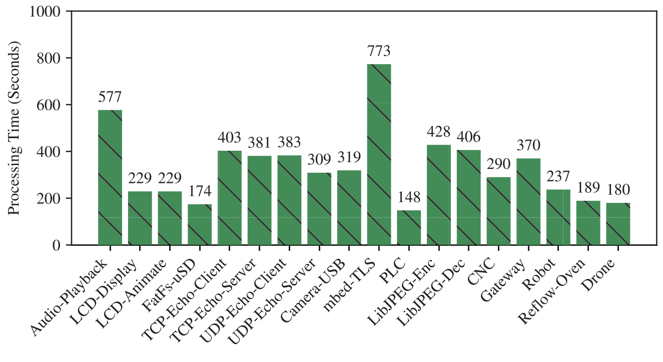


Fig. 5. The proposed framework processing time for the benchmark suite.

TABLE 5
Comparison of Fuzzing Original and Sanitized Bare-Metal Firmware in HAL-Fuzz [9] and $P^2$IM [10]

| Memory Corruptions | HAL-Fuzz | HAL-Fuzz with Sanitization | $P^2$IM | $P^2$IM with Sanitization |
|---|---|---|---|---|
| Audio-Playback | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| LCD_Display | (S) | (S, O, D, U) | (S, U) | (S, O, D, U) |
| LCD_Animate | (S) | (S, O, D, U) | (S, U) | (S, O, D, U) |
| FatFs_uSD | ✗ | (S, O, D, U) | (S) | (S, O, D, U) |
| TCP_Echo_Client | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| TCP_Echo_Server | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| UDP_Echo_Client | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| UDP_Echo_Server | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| Camera-USB | (U) | (S, O, D, U) | (S) | (S, O, D, U) |
| mbed-TLS | (S, U) | (S, O, D, U) | (U) | (S, O, D, U) |
| PLC | (U) | (S, O, D, U) | ✗ | (S, O, D, U) |
| LibJPEG_Encoding | ✗ | (S, O, D, U) | (S) | (S, O, D, U) |
| LibJPEG_Decoding | ✗ | (S, O, D, U) | (U) | (S, O, D, U) |
| CNC | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| Gateway | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| Robot | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| Reflow Oven | ✗ | (S, O, D, U) | ✗ | (S, O, D, U) |
| Drone | (U) | (S, O, D, U) | ✗ | (S, O, D, U) |

*Each entry represents detected Stack-based buffer overflow (S), Off-by-one (O), Double Free (D), and Use-after-free (U) vulnerabilities in the target firmware.*

As we already mentioned, simply counting the number of crashes or found vulnerabilities to evaluate the fuzzing effectiveness is too coarse-grained. As a result, in our evaluation, we make distinctions between reaching, triggering, and detecting a vulnerability to evaluate IoT fuzzer effectiveness. A reached vulnerability is the one that the fuzzer reaches its location in the executed path without necessarily triggering a fault. For example, consider that we have a buffer overflow vulnerability in our firmware binary; the fuzzer can generate an input to reach that vulnerability but cannot overflow the buffer. While, a fuzzer triggers a vulnerability by satisfying its triggering conditions and causes a faulty state. However, this faulty state can be not detected by the fuzzer. This is the case for most of the IoT fuzzer that do not sanitize the firmware and face silent memory corruption vulnerabilities. In our benchmark framework, we evaluate the IoT fuzzers regarding reaching, triggering, and detecting a vulnerability by IoT fuzzers and improve their detection capability with our sanitization framework.

The proposed framework instrumented and sanitized vulnerable firmware binaries from our benchmark. As illustrated in Table 5, the framework made all faulty states caused by memory corruption vulnerabilities detectable. More precisely, it triggers crashes upon faults caused by stack-based buffer overflow, stack off-by-one, double free, and use-after-free, allowing HAL-Fuzz and $P^2$IM detect them. It is worth mentioning that for the un-sanitized firmware that fuzzers could not detect the vulnerabilities, since we already obtained the triggering input by symbolic execution and fed the fuzzer with them, the vulnerabilities are reached and triggered but not detected by the fuzzers. For example, as no memory protection is provided in our bare-metal device, most firmware containing stack-based buffer overflow vulnerability continue the execution as usual unless they were sanitized with our proposed framework, which spots the out-of-bounds write. Furthermore, it automatically identified all the locations and types of memory corruption vulnerabilities correctly which helps security analysts to patch them.

Note that, different fuzzing results between original HAL-Fuzz and $P^2$IM are due to the different approaches that they used for their fuzzing procedure such as security policies used for detecting crashes and differences between AFL and AFL-Unicorn fuzzers used by these frameworks. However, we refer readers interested in a detailed explanation on the different aspects of HAL-Fuzz and P2IM to their original papers since the aim of this work is not to compare these two fuzzing frameworks together, and we demonstrated these results to show that how the effectiveness of the state-of-the-art IoT fuzzers can be improved if they will be combined with our proposed static binary sanitization approach.

## 6 Related Work

In this section, we systematically analyze related work that are both complementary and orthogonal to fuzzing embedded devices.

### 6.1 IoT Fuzzing & Firmware Re-Hosting

Nowadays, fuzzing becomes one of the most effective vulnerability detection methods for IoT devices. For instance, Zheng *et al.* proposed Firm-AFL [18], built atop AFL [8] and Firmadyne [46], in order to fully emulate embedded firmware and perform greybox fuzzing on them. Similarly, FirmFuzz [19] provides an emulation mechanism for dynamic analysis and fuzzing embedded firmware. However, instead of bare-metal firmware, these solutions support only Linux-based firmware images that are closer to general-purpose computers than low-power IoT devices. Additionally, FirmFuzz relies on some manual validation by the analyst that limits the scalability. IoTFuzzer [17] is a blackbox fuzzer aiming at finding memory corruption vulnerabilities in IoT devices without access to their firmware images and through their companion mobile applications. Nevertheless, it is a "hardware-in-the-loop" approach and requires the presence of the IoT device, limiting the ability to scale fuzzing. HAL-Fuzz [9] is a greybox fuzzer that is built on top of HALucinator emulator [45]. HAL-Fuzz utilizes AFL-Unicorn [47] to perform fuzzing process.

Emulation, also known as firmware re-hosting, takes the firmware out of its original execution environment and provides the ability for executing it at scale through the use of general purpose computers. There have been a number of firmware re-hosting methods [48], [49] that provide a hybrid execution environment by forwarding peripheral operations to physical target devices while running firmware on a standard emulator (e.g., QEMU). However, such re-hosting methods have heavy hardware dependence, and only one fuzzing session is possible per-device, which essentially limit the scale of the fuzzing. Furthermore, resetting the firmware execution, which frequently happens to generate a clean state for the next test case, can incur a significant overhead due to the need for a full reboot of the physical device. Along similar lines, Pretender [50] and $P^2IM$ [10] provide a fully emulated environment for IoT firmware by directly modeling the MMIO peripherals, while HALucinator [45] instead utilizes HAL libraries for firmware re-hosting.

## 6.2 Binary Rewriting

Binary rewriting refers to the process of modifying one binary into another by optionally inserting one or more new instruction, either statically or dynamically, to provide new features or behaviors while maintaining original functionality. Dynamic binary rewriting mechanisms [16], [51] transform stripped binaries that are loaded into memory while they are executing. However, they are not practical for instrumenting bare-metal firmware due to the high performance overhead and special software/hardware requirements.

There are a number of static mechanisms that transform binaries before execution. These mechanisms differ from each other in how they transform binaries without breaking their functionality and semantics. Trampoline-based rewriters such as Bistro [52] and STIR [53] replace original instructions at a target instrumentation point with a new branch instruction. This new branch instruction redirects application control flow from the original location to the trampoline block containing both the added instrumentation logic and the original instructions replaced by the branch instruction. Such rewriters are able to preserve application semantics after instrumentation, at the cost of considerable performance and memory penalties.

Uroboros [29] and Ramblr [28] present a set of heuristics to convert a binary into their own internal representations and perform instrumentation on those. However, heuristics-based approaches suffer from false positives and negatives that result in broken reassembled binary. RetroWrite [54] and Egalito [55] instrument executable binaries by leveraging relocation information which is only available in position independent codes. Unfortunately, this is not an applicable solution for bare-metal firmware that are statically linked. Multiverse [56] proposes a new disassembling technique by disassembling instructions from every offset of code section to create a superset of all possible disassemblies. Multiverse binary rewriter is built upon this disassembler to instrument all superset instructions. As pointed out by Miller *et al.* [57], superset disassembly technique incurs a substantial code size overhead (763% on SPECint 2006 benchmarks). In addition, experimental results [55] show that Multiverse does not support statically linked binaries.

All the above binary rewriters only target x86 architecture. RevARM [30] is the only static binary rewriter that provides support for the ARM architecture. However, RevARM is designed and developed for instrumenting mobile applications, not bare-metal firmware which have significant resource constraints. Furthermore, RevARM uses unsound heuristic-based approach for rewriting binaries statically. Indeed, RevARM leverages a similar approach to Uroboros [29] for recognizing pointer-like data and identifying code pointers, an approach which is proved to be unsound in [28] and [54]. Moreover, RevARM instruments applications at a higher-level intermediate representation (IR). Dinesh *et al.* [54] noted that lifting disassembly to a higher-level intermediate language requires precise modeling of the instruction set architecture (ISA), which is an error-prone process.

## 6.3 Fault Detection

Muench *et al.* [3] proposed a fault detection method in embedded systems by integrating a black-box fuzzer Boofuzz [7] with a set of six heuristics such as heap object tracking in order to detect faulty states caused by memory corruptions. However, these heuristics' effectiveness relies on various information, including memory accesses, memory mappings, executed instructions, register state, and (de) allocation functions. Such information must be extracted from applying reverse engineering techniques and additional annotations provided manually by the analyst, adding both imprecision and complexity. Furthermore, experimental results show that applying heuristics for fault detection still has false positives and false negatives.

## 6.4 Fuzzing Benchmark

Google FuzzBench [58] is a fuzzer benchmarking service that considers coverage profiles as a metric for evaluating fuzzers' effectiveness. However, this metric is inappropriate for evaluating fuzzers' bug-finding capability [5], [59]. LAVA-M test suite [60] creates a ground-truth corpora which are used as evaluation metrics for fuzzing performance. LAVA injects vulnerabilities in different execution paths and, using taint-analysis techniques, provides inputs to trigger them. Magma [59] is an open ground-truth benchmark containing real vulnerable applications for evaluating and comparing fuzzing mechanisms' performance. However, all these benchmarks contain vulnerable applications that are developed for general-purpose computers. Therefore, such benchmark suites are not practical for evaluating IoT fuzzers.

## 7 Discussion

While our approach improves the feasibility of fault detection in fuzzing bare-metal firmware, there are still some challenges for future improvements. In this section, we discuss the existing limitations in the current design and explore how they could be handled in the future.

## 7.1 Supported Firmware

Our prototype implementation relies on static binary instrumentation to sanitize bare-metal firmware. As such, it is unable to instrument code that is generated dynamically.

Indeed, dynamically generated code can only be instrumented using dynamic instrumentation methods. Also, our prototype implementation is presently compatible with widely used ARM based bare-metal devices. However, since our proposed approach is platform-independent, supporting other architecture requires a small extra engineering effort.

### 7.2 Sanitization

Our approach does not sanitize global variables since performing arbitrary transformations on the layouts of data sections requires recovering semantic information that was lost during compilation. Indeed, information about global data section layouts is lost at the binary level, which makes it impossible to insert a fully precise spatial memory safety sanitizer using binary instrumentation. However, it is worth noting that in comparison to the number of heap or stack allocations, the number of global objects in a firmware is fixed and relatively small. Therefore, in comparison to source-based sanitizers like AddressSanitizer, our approach may miss a fixed number of faults if a global overflows into adjacent memory.

### 7.3 Fuzzing Approach

This paper aims to propose a static binary sanitizer mechanism for improving IoT fuzzers capabilities in observing and identifying memory corruptions. However, other aspects of fuzzing such as input generation process and code coverage are orthogonal to our domain and do not affect our work.

### 7.4 Vulnerability Injection Location

Our benchmark framework can only inject vulnerabilities inside input functions. This is a simplification we consider to avoid the path explosion problem during the triggering input generation by symbolic execution. Indeed, this problem is inherited from symbolic execution analysis and associated constraint solvers. However, despite the increasing amount of research [61], [62], [63] dedicated to this topic, the ability to cover execution paths effectively is still an open problem and orthogonal to our work. However, this does not totally limit the effectiveness of our benchmark to control how deep into execution the vulnerabilities are injected. Specifically, protecting the vulnerability with conjunctive conditional statements leads to the creation of dependent basic blocks, i.e., equivalent to nested `ifs`.

In conclusion, our benchmark framework goal is to evaluate the bug finding capability of IoT fuzzers and raise awareness about the impact of binary sanitization when encountering silent memory corruptions, and it is limited to evaluate the coverage exploration capabilities of IoT fuzzers. However, we provide the building blocks to solve this problem, and will hopefully stimulate new research in this important direction.

### 7.5 Firmware Emulation

In order to effectively take advantage of fuzzing, contemporary general purpose computers execute multiple instances of the same application at scale through the use of multiprocessing or virtualization. Indeed, throughput is a key factor for the effectiveness of fuzzing methods [18].

However, due to the lack of standardized hardware, diverse firmware, and opaque functionality, bare-metal devices present unique challenges to execute firmware in emulated or virtualized environments. More precisely, due to the rise of highly-integrated chip designs, bare metal firmware interact with an enormous selection of external peripherals customized by the MCU vendors. Therefore, it is often required to design and implement a specially customized emulator for a new bare metal firmware.

## 8 Conclusion

In this paper, we distill the key challenges in fuzzing bare-metal devices and emphasize the necessity of observing faulty states caused by memory corruptions. We proposed a novel static binary instrumentation solution to automatically sanitize bare-metal firmware binaries with memory access validations. Using static binary sanitization, we achieve not only fault detection capability in fuzzing bare-metal firmware, but also the identification of the location and the root cause of memory corruptions. Moreover, we proposed a ground-truth fuzzing benchmark that enables uniform IoT fuzzer evaluation and comparison.

Combining our sanitization mechanism with state-of-the-art IoT fuzzers enables us to identify previously undiscovered memory corruptions in the proposed benchmark. Also, the evaluation shows that our approach is able to automatically identify the type of discovered memory corruptions where previous approaches fail. Our framework and the benchmark suite are available as open-source at https://github.com/pwnforce/uSBS

## Acknowledgments

We are grateful to anonymous reviewers for assisting us with their helpful comments and criticisms.

## References

[1] Ericsson, "Internet of Things forecast," 2019. Accessed: Feb. 2021. [Online]. Available: https://www.ericsson.com/en/mobility-report/internet-of-things-forecast

[2] Google project zero, "Over the air: Exploiting broadcoms Wi-Fi stack," 2017. Accessed: Feb. 2021. [Online]. Available: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[3] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[4] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proc. 18th Conf. USENIX Secur. Symp.*, 2009, pp. 67–82.

[5] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2018, pp. 2123–2138.

[6] M. Salehi, D. Hughes, and B. Crispo, "$\mu$SBS: Static binary sanitization of bare-metal embedded devices for fault observability," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, 2020, pp. 381–395.

[7] J. Pereyda, "BooFuzz source code repository," 2016. Accessed: Feb. 2021. [Online]. Available: https://github.com/jtpereyda/boofuzz

[8] M Zalewski, "American fuzzing lop (AFL)," 2014. Accessed: Feb. 2021. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[9] A. Clements *et al.*, "HALucinator-fuzzer source code repository," 2020. Accessed: Feb. 2021. [Online]. Available: https://github.com/ucsb-seclab/hal-fuzz

[10] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proc. 29th USENIX Secur. Symp.*, 2020, Art. no. 70.

[11] D. Song *et al.*, "SoK: Sanitizing for security," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 1275–1295.

[12] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 17–30.

[13] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 309–318.

[14] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[15] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *Proc. Int. Symp. Code Gener. Optim.*, 2011, pp. 213–223.

[16] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 89–100.

[17] J. Chen *et al.*, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[18] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1099–1114.

[19] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT firmware introspection and analysis," in *Proc. 2nd Int. ACM Workshop Secur. Privacy Internet-of-Things*, 2019, pp. 15–21.

[20] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 711–725.

[21] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.

[22] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 845–860.

[23] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proc. 9th Work. Conf. Reverse Eng.*, 2002, pp. 45–54.

[24] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proc. 20th Int. Conf. Comput. Aided Verification*, 2008, pp. 423–427.

[25] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 583–600.

[26] M. Salehi, D. Hughes, and B. Crispo, "MicroGuard: Securing bare-metal microcontrollers against code-reuse attacks," in *Proc. IEEE Conf. Dependable Secure Comput.*, 2019, pp. 1–8.

[27] C. H. Kim *et al.*, "Securing real-time microcontroller systems through customized memory view switching," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[28] R. Wang *et al.*, "Ramblr: Making reassembly great again," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[29] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 627–642.

[30] T. Kim *et al.*, "RevARM: A platform-agnostic ARM binary rewriter for security applications," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, 2017, pp. 412–424.

[31] Capstone, "Capstone: The ultimate disassembler framework," 2020. Accessed: Feb. 2021. [Online]. Available: http://www.capstone-engine.org/

[32] E. Bendersky, "Pyelftools: Parsing ELF and DWARF in Python," 2012. Accessed: Feb. 2021. [Online]. Available: https://github.com/eliben/pyelftools/

[33] Quarkslab, "Quarkslab Lief project," 2020. Accessed: Feb. 2021. [Online]. Available: https://lief.quarkslab.com/

[34] Pwntools, "CTF framework and exploit development library," 2020. Accessed: Feb. 2021. [Online]. Available: https://github.com/Gallopsled/pwntools/

[35] STM32Cube MCU packages. Accessed: Sep. 2021. [Online]. Available: https://www.st.com/en/embedded-software/stm32cube-mcu-mpu-packages.html

[36] Quad-copter drone source code. 2017. Accessed: Sep. 2021. [Online]. Available: https://github.com/heethesh/eYSIP-2017_Control_and_Algorithms_development_for_Quadcopter

[37] CNC GRBL STM32F4 source code. 2016. Accessed: Sep. 2021. [Online]. Available: https://github.com/deadsy/grbl_stm32f4

[38] Firmata Library. 2018. Accessed: Sep. 2021. [Online]. Available: https://github.com/firmata/arduino

[39] Self-balancing robot source code. 2020. Accessed: Sep. 2021. [Online]. Available: https://github.com/mbocaneg/Inverted-Pendulum-Robot

[40] Reflow oven source code. 2012. Accessed: Sep. 2021. [Online]. Available: https://github.com/rocketscream/Reflow-Oven-Controller

[41] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2014, pp. 389–391.

[42] Y. Shoshitaishvili *et al.*, "SoK: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.

[43] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice – Automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[44] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[45] A. Clements *et al.*, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *Proc. 29th USENIX Secur. Symp.*, 2020, Art. no. 68.

[46] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[47] N. Voss, "AFL-unicorn," 2017. Accessed: Feb. 2021. [Online]. Available: https://github.com/Battelle/afl-unicorn

[48] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014.

[49] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *Proc. 9th USENIX Conf. Offensive Technol.*, 2015, Art. no. 7.

[50] E. Gustafson *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *Proc. 22nd Int. Symp. Res. Attacks Intrusions Defenses*, 2019, pp. 135–150.

[51] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200.

[52] Z. Deng, X. Zhang, and D. Xu, "BISTRO: Binary component extraction and embedding for software security applications," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2013, pp. 200–218.

[53] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 157–168.

[54] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *Proc. 41th IEEE Symp. Secur. Privacy*, 2020, pp. 1497–1511.

[55] D. Williams-King *et al.*, "Egalito: Layout-agnostic binary recompilation," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 133–147.

[56] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[57] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 1187–1198.

[58] Google, "Fuzzbench," 2020. Accessed: Feb. 2021. [Online]. Available: https://google.github.io/fuzzbench/

[59] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, 2020, Art. no. 49.

[60] B. Dolan-Gavitt *et al.*, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 110–121.

[61] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 1083–1094.

[62] D. Engler and D. Dunbar, "Under-constrained execution: Making automatic code destruction easy and scalable," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 1–4.

[63] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 49–64.

**Majid Salehi** received the MSc degree from the Sharif University of Technology, in 2016. He is currently a PhD researcher with KU Leuven Computer Science Department, where he is a member of Imec-DistriNet Research Group. His research interests include Internet of Things (IoT) security and issues concerning memory-based attacks in bare-metal embedded devices.

**Luca Degani** received the master's degree with a thesis on evaluation metrics of bare-metal IoT firmware fuzzing. He is currently working toward the PhD degree in computer science with the University of Trento, Italy, where he is working on IoT systems security. His research interests include vulnerability detection and software testing.

**Marco Roveri** received the PhD degree in computer science from the University of Milano, Italy, in 2002. He is currently an associate professor with the Information Engineering and Computer Science Department, University of Trento, Italy. He was a senior researcher with the Embedded Systems Unit of Fondazione Bruno Kessler, Trento, and before a researcher with the Automated Reasoning Division, Istituto Trentino di Cultura, Trento. His research interests include automated formal verification of hardware and software systems, formal requirements validation of embedded systems, model based predictive maintenance, and automated model based planning, and application of such techniques in industrial settings.

**Danny Hughes** is currently a professor with the Department of Computer Science, KU Leuven, Belgium, where he is a member of Imec-DistriNet (Distributed Systems and Computer Networks) Research Group and leads Networked Embedded Software taskforce. His current research interests icnlude distributed software systems and the Internet of Things.

**Bruno Crispo** (Senior Member, IEEE) received the PhD degree from Cambridge University, U.K. He is currently a full professor of computer science with the University of Trento, Italy, and a visiting professor with KU Leuven, Belgium. His research interests include IoT security, network security, web security, biometric authentication, and access control. He is an associate editor for the *ACM Transactions on Privacy and Security*.