μSBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability

Majid Salehi
imec-Distrinet, KU Leuven
majid.salehi@kuleuven.be

Danny Hughes
imec-Distrinet, KU Leuven
danny.hughes@kuleuven.be

Bruno Crispo
imec-Distrinet, KU Leuven
Trento University, Italy
bruno.crispo@unitn.it

Abstract

A large portion of the already deployed Internet of Things (IoT) devices are bare-metal. In a bare-metal device, the firmware executes directly on the hardware with no intermediary OS. While bare-metal devices increase efficiency and flexibility, they are also subject to memory corruption vulnerabilities that are regularly uncovered. Fuzzing is an effective and popular software testing method to discover vulnerabilities. The effectiveness of fuzzing approaches relies on the fact that memory corruption faults, by violating existing security mechanisms such as MMU, are observable, thus relatively easy to debug. Unfortunately, bare-metal devices lack such security mechanisms. Consequently, fuzzing approaches encounter silent memory corruptions with no visible effects making debugging extremely difficult. This paper tackles this problem by proposing μSBS, a novel approach that, by statically instrumenting the binaries, makes memory corruptions observable. In contrast to prior work, μSBS does not need to reverse engineer the firmware. The approach is practical as it does not require a modified compiler and can perform policy-based instrumentation of firmware without access to source code. Evaluation of μSBS shows that it reduces security analyst effort, while discovering the same set of memory error types as prior work.

1 Introduction

Recent years have witnessed the proliferation of Internet of Things (IoT) devices into nearly every aspect of our lives. According to a recent Gartner report [5], the number of connected IoT devices is expected to exceed the total number of humans by 2020. A large portion of these devices are bare-metal with the firmware running directly on the hardware. This approach can deliver energy-efficiency, extensible connectivity, and adequate computing power. However, most of these firmware are implemented in type-unsafe languages such as C, C++, or Objective-C, that are prone to memory corruption vulnerabilities such as buffer overflows. This creates a very large attack surface in the IoT ecosystem.

Given the limited resources of bare-metal devices, traditional mitigation mechanisms for memory corruption vulnerabilities such as Control Flow Integrity (CFI) [24,44], Address Space Layout Randomization (ASLR), and security policy reinforcement [57] are typically infeasible [16]. More importantly, many IoT devices today work in a real-time environment and must remain responsive to external stimuli (e.g., a health-care system, or a safety system in a car). These systems cannot accommodate the high run-time overhead incurred by most mitigation mechanisms. This highlights the importance of performing a vulnerability discovery process before the firmware is released, thus at testing time.

Fuzz-testing or Fuzzing is a testing solution for finding bugs and vulnerabilities. Fuzzing methods execute the application with randomly generated inputs and wait for vulnerability-exposing behaviors such as crashing or hanging. This behavior is the visible consequences of faulty states triggered by deployed security mechanisms such as Memory Management Unit (MMU). Unlike general purpose computers, bare-metal devices often lack such mechanisms due to their cost sensitivity and resource constraints. Accordingly, fuzzing bare-metal devices is extremely challenging to debug since memory corruptions may trigger no observable behaviors and thus cannot be discovered through fuzzing.

To address this problem, Muench et al. [42] integrated the black-box fuzzer Boofuzz [2] with a set of heuristics to recognize faults due to memory corruptions[1], but experimental results show it has still false positives and false negatives. Even more important, the proposed heuristics rely on information extracted from applying reverse engineering techniques and additional annotations provided manually by the analyst, all activities that are challenging and time consuming. Furthermore, they need to be applied for each new firmware under analysis.

Sanitizers [51], can be combined with fuzzing methods in order to make faulty states observable. Sanitizers instrument applications with memory check instructions to monitor all

[1] In the interests of brevity we refer to a faulty state caused by memory corruption simply as a fault for the rest of the paper.

reads and writes during application execution. There are sanitizers that operate at the source code or compilation level, such as AddressSanitizer [46], while others, as Valgrind's Memcheck [48] that operate on machine code. Considering that the majority of bare-metal firmware are not open-source, source-based sanitizers are not the best choice in our context. Binary sanitization could be done either statically or dynamically. Dynamic binary sanitizers allow instrumentation of an application at runtime. However, such techniques are not widely deployable on the bare-metal devices mainly due to the high performance penalties and special software/hardware requirements. On the other hand, static binary sanitizers introduces lower overhead by instrumenting application binary statically. Unfortunately, at the time of writing none of the current binary sanitizers provides support for bare-metal devices.

This paper presents μSBS, a novel approach that, by statically instrumenting bare-metal firmware binaries, makes memory corruptions observable. μSBS provides a static binary instrumentation method and uses it for instrumenting memory instructions (i.e., sanitization). μSBS allows to embed a given memory safety policy and to monitor all memory accesses, triggering observable warnings when a violation to the policy occurs. In summary, the paper makes following contributions:

- We present μSBS, the first static binary sanitizer for bare-metal firmware. It avoids the complex and tedious work of reverse engineering firmware binaries.

- Using μSBS, we make memory corruption faults observable also on bare-metal devices, thus facilitating their debugging.

- We developed a fully functional prototype of μSBS for the ARM architecture which is the most widely used architecture in IoT devices. To foster further research, we make our μSBS prototype available open source.

- We evaluated the effectiveness of μSBS in catching the same classes of memory faults of prior work. We assess the feasibility of μSBS by instrumenting 11 real-world firmware binaries. Evaluation results show that μSBS correctly instruments all of the firmware binaries with reasonable execution over-head and size expansion.

2 Background and Motivation

In this section, we present a brief overview of 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 faults observability on such architectures.

2.1 Memory Corruptions and Fuzzing

Low-level systems software such as firmware is typically written in the C or C++ languages due to their efficiency and capability to fully control the underlying hardware. In such programming languages, developers must ensure that every memory access is valid, that no situation leads to the de-referencing of invalid pointers. However, in practice, developers frequently fail to meet these responsibilities and cause memory bugs that can be exploited by an attacker to alter the application behavior or even taking full control over the software stack.

In testing the security of such application, security analysts hardly have access to the source code. Fuzzing is one of the most effective testing methodologies to find memory corruption vulnerabilities in Commercial Off-The-Shelf (COTS) applications. Fuzzing executes the application binary file with random inputs to look for unexpected application behavior such as crashes that are immediate consequences of faulty states. The ability of observing such crashes is the prerequisite for fuzzing to work. In general purpose computer systems, equipped with OS security mechanisms and hardware features such as stack canaries, Data Execution Prevention (DEP), Memory Management Unit (MMU), and Memory Protection Unit (MPU), memory violations trigger a crash upon a fault. Possible ways to observe such crashes are: (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.

2.2 Bare-metal Embedded Devices

Among different classes of embedded devices, bare-metal devices are designed for low cost and low power operation. Such devices are deployed in many application areas ranging from automotive and industrial control systems to medical devices. Bare-metal devices execute a single statically linked binary firmware providing a specific application logic as well as system functionality without the use of an underlying abstraction such as an operating systems. However, it is challenging for bare-metal devices to support security properties in practice, due to limited energy, memory and computing resources. For example, this class of devices rarely provide a Memory Management Unit (MMU) and firmware modules have access to the entire shared memory space in a privileged mode. Therefore, compromising one firmware module gives an attacker arbitrary read/write access to the whole system with no observable side-effects. Unrestricted read/write primitive enables the attacker to redirect the control-flow of firmware or directly overwrite sensitive data.

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.

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 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 Fault Observability in Bare-metal Devices

Contemporary general purpose computers have plenty of mechanisms that makes faulty states observable (e.g., segmentation faults caused by an MMU). Most bare-metal devices, instead, do not have such mechanisms due to their limited I/O capabilities and architecture. In fact, most memory corruptions events are *silent* and do not lead to an immediate crash of the firmware or any observable event. Thus the firmware can continue the execution with no visible effect or it will lead eventually to a crash (i.e., I/O error) that is however very difficult to debug. It is challenging to infer if the crash was due to an early memory violation or to an I/O error.

Motivating Example. To better understand the problem, we use a popular bare-metal firmware, Broadcom Wi-Fi SoC as a motivating example. This firmware is present in both mobile devices and Wi-Fi routers for handling the lower layers of Wi-Fi and Bluetooth protocols. The Broadcom Wi-Fi SoC executes on ARM Cortex-R processor. As reported by Google Project Zero [6] in CVE-2017-0561 [4], the firmware has a remote code execution vulnerability that enables a remote attacker to execute arbitrary code and escalate to control over the entire system. In the code snippet shown in Listing 1, SoC firmware performs a *memcpy* into the allocated memory object *buffer*, using the *ft_ie* length field. Since the *ft_ie* length field is not verified prior to the copy, this allows an attacker to exceed the *buffer* and trigger a buffer overflow.

```
uint8_t* buffer = malloc(256);
...
uint8_t* linkid_ie = bcm_parse_tlvs(..., 101);
memcpy(buffer, linkid_ie, 0x14);
...
uint8_t* ft_ie = bcm_parse_tlvs(..., 55);
memcpy(buffer + 0x18, ft_ie, ft_ie[1] + 2);
```

Listing 1: A remote code execution vulnerability in the Broadcom Wi-Fi firmware could enable a remote attacker to execute arbitrary code within the context of the Wi-Fi SoC.

As a proof of concept we exploited this vulnerability similarly to [6] and sent malicious inputs that triggered the buffer overflow vulnerability. Nonetheless, the firmware did not crash and continued to function normally with no observable side-effects. This is mainly due to the fact that Broadcom Wi-Fi SoC lacks all basic memory protection mechanisms

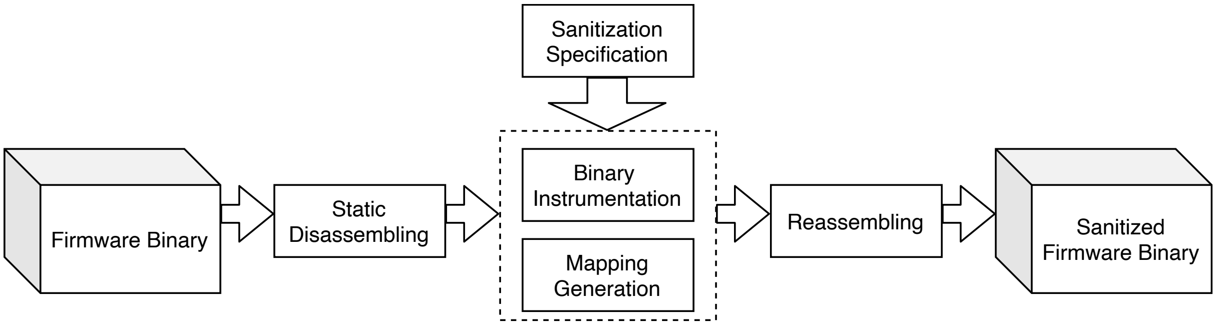


Figure 1: Workflow of μSBS.

including access permission protection by means of either MMU, MPU or more advance protections like stack canaries.

There exist several approaches to make faulty states observable. Sanitizers [51] monitor the actual execution of an application in order to observe faulty states as they happen. They enforce *spatial* memory safety by detecting dereferences of pointers that do not target their intended referent, or enforce *temporal* memory safety by detecting dereferences of pointers that target a referent that is no longer valid. Sanitizers implement memory safety policies by embedding inlined reference monitors (IRM) into the application through instrumentation. IRMs mediate and monitor every memory accesses and memory object (de)allocations instructions. IRMs can be embedded at either source code or binary level. Source-based sanitizers [30, 33, 46, 53, 60] are not widely deployable on bare-metal devices due to the unavailability of their firmware source code since they are often proprietary.

Binary images of bare-metal firmware are often available to the analyst since they can be acquired by directly extracting from the physical device using debugging port (e.g., JTAG interface) or downloading and unpacking update packages available on many vendors websites. Thus, binary sanitization [23, 34, 48] is the only viable option for enforcing memory safety in bare-metal firmware. Dynamic binary sanitizers read application code, instrument it, and translate it to machine code while the application executes. However, these approaches do not generate a standalone instrumented binary and sanitization process has to be done again each time the application executes. In addition, they have significant run-time and space overhead which is a critical problem for bare-metal devices which have limited processing power and a small memory space. This overhead can essentially be attributed to the dynamic translation process. This issue can be addressed by instrumenting applications statically using a binary sanitizer, which we believe is a promising solution for our requirements.

Until now, there is no binary sanitizer for bare-metal firmware. Based on this limitation of previous work, we propose a novel and tailored automated method for making faults observable in bare-metal devices.

3 Static Binary Sanitization for Bare-metal Devices

Figure 1 illustrates a high-level overview of our approach, with the different components and their interactions. There are three main phases: the *static disassembling*, the *binary instrumentation*, and the *reassembling*.

The first step of our μSBS workflow is *static disassembling* (§ 3.1). It disassembles the raw binary and decodes instructions using a linear disassembly method. The second phase is *binary instrumentation* (§ 3.2) that instruments the firmware binary based on the sanitization specification. Sanitization specification (§ 3.3) contains instrumentation information determining what instructions will be inserted or replaced in order to embed IRMs and monitor every memory access. In other words, μSBS 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 warning close to the location of the bug to guide the follow-up security analysis to find out the root cause of the fault. The last step is the *reassembling*, that takes the instrumented assembly code and reassembles it as a working binary using off-the-shelf assemblers. In the following sections, we describe the most important and challenging aspects of μSBS design.

3.1 Static Disassembling

Disassembly is referred as the process of parsing executable region of binary file from beginning to the end and decoding all encountered bytes into their raw textual representation. There are two popular types of disassembly approaches: linear sweep and recursive traversal disassemblers [19, 37, 45]. Linear sweep decodes all encountered bytes as instructions by sweeping the entire code section. Recursive traversal disassembles instructions following control flow transfers (e.g., jumps and calls). It is challenging to correctly and completely disassemble arbitrary code. This is mainly due to the fact that, in hand-written assembly and modern compilers, code and data can be interleaved and there is no syntactic distinction whereby the disassembler may distinguish them. However,

```
#include<stdio.h>
void printer_ver1 (float var) {
     printf("%.10f\n", var);
}
void printer_ver2 (float var) {
     printf("%.50f\n", var);
}
void (*printers[2])(float) = {printer_ver1,
    printer_ver2};

int main (void)
{
     static float var = 4e-34;
     static int (**ptr)(float)= &printers;
     (*(ptr+1))(var);
     return 0;
}
```

(a) Source code of running example.

```
0x804816c <printer_ver2>:
...
0x8048198:     bl    8048688 <printf>


0x80481b0 <main>:
...
0x80481bc:     movw  r3, #64668 ; 0xfc9c
0x80481c0:     movt  r3, #1
0x80481c4:     ldr   r3, [r3]
0x80481c8:     add   r3, r3, #4
0x80481cc:     ldr   r2, [r3]
0x80481d0:     movw  r3, #64672 ; 0xfca0
0x80481d4:     movt  r3, #1
0x80481d8:     ldr   r3, [r3]
0x80481dc:     mov   r0, r3
0x80481e0:     blx   r2
```

(b) Partial assembly code of running example.

```
Hex dump of section .data:
0x0001fc90 00000000 28810408 6c810408 94fc0100
0x0001fca0 3dec0408 00000000 00000000 94ff0100
```

(c) Hexdump of .data section.

Figure 2: A running example that covers three main challenges of binary instrumentation and illustrates four general classes of references (C2C, C2D, D2D, D2C) in assembly code.

Andriesse et al. [18] noted that such cases are exceedingly rare and disassemblers achieve close to 100% accuracy for instruction disassembly from compiler-generated binaries. Therefore, we applied a linear sweep disassembly algorithm to our evaluation set.

3.2 Binary Instrumentation

The μSBS *binary instrumentation* component takes the disassembled file and sanitization specification as inputs with the aim of statically inserting a number of memory check instructions to catch memory corruption vulnerabilities. However, binary instrumentation introduces challenges that are not present when modifying source code. Specifically, when instructions are inserted or removed at the source code level, the compiler will redo the linking process to rearrange code and data in memory. In binaries, inserting or removing instructions causes addresses to change and breaks the binary file due to the lack of linkage information. The symbol and relocation information, that is used in the linking process to ensure that application elements can correctly refer to each other, are discarded by the compiler once finished. In the following, we provide the details of practical challenges in designing μSBS *binary instrumentation* component and our solutions tackling them.

The core process of instrumenting binaries is the ability to relocate any binary code without any relocation and metadata information. There are three main challenges in relocation procedure to avoid breaking the binary file. To describe these challenges clearly, Figure 2 shows a running application alongside its disassembly and the hex dump of its data section. This application declares a float variable *var* with an initial value of 4e-34 and prints that with two different formats (i.e., *printer_ver1* and *printer_ver2*). 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. In our application, the compiler stores 3dec0408 in data section as the binary representation of *var*. Since our application has a code section with memory addresses ranging from 0x8000000 to 0x8100000, this immediate value can be considered as a pointer (0x804ec3d) on little-endian machines; this will irremediably corrupt the image if we update it after binary instrumentation.

The second challenge is relocating static addresses after instrumentation. For example, as illustrated in our application disassembly, the *printer_ver2* function calls the *printf* function with static address 0x8048688. It is clear that the insertion of instructions into, or removal of instructions from disassembly code can break this static address. 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. As shown in our application disassembly, the reference target *r2* is computed dynamically.

To tackle these challenges, we categorize all references into four general classes: Code-to-Code (C2C), Code-to-Data (C2D), Data-to-Code (D2C), and Data-to-Data (D2D). In our application, the *ldr r3, [r3]* instruction accesses to the address 0x0001fc9c in order to retrieve the *printers* array address from data section (C2D). 94fc0100 hex value at address 0x0001fc9c is a pointer to the *printers* array in data section (D2D). The second element (6c810408) of the *printers* array points to the *printer_ver2* function in code section (D2C), which calls the *printf* function at address 0x8048688 (C2C).

C2D and D2D references. Due to the fact that there is no need to perform instrumentation on the original data space, we can preserve the starting addresses of data sections intact. By doing so, we may easily ignore and handle C2D and D2D references.

C2C and D2C references. Since insertion of instructions causes stretched code space, μSBS adds a new expanded code section (*.newsec*) at a new entry point. As demonstrated in Algorithm 1, μSBS iterates all disassembled instructions and rewrites them intact in the *.newsec* section. Also, in the meanwhile, μSBS performs instrumentation and inserts new memory check instructions in the *.newsec* section.

μSBS adjusts all branch instructions target addresses while rewriting them in the *.newsec* section. Each direct branch instruction with an immediate operand can easily point to the new address by changing its offset statically. However, indirect branch instructions have multiple possible target addresses and therefore needs some sort of target-prediction mechanism. Unlike many prior efforts [36, 54, 55], we observe that while it is challenging to statically identify targets of indirect branch instructions, we can instead perform a dynamic lookup at runtime. It is mainly due to the fact that the precise target addresses are known at runtime. Specifically, we provide a mapping table from the old code section to *.newsec*. By doing so, we can modify each indirect branch instruction to search for the new target address in the mapping table after the old target address has been computed at runtime.

To generate the mapping table, it is first required to know each instruction size that is present in the *.newsec* section. Therefore, we record any changes to instructions and sizes while rewriting them in the *.newsec* section. More specifically, we generate a mapping table from each address in the old code section to the size of the corresponding rewritten bytes in the *.newsec* section. Afterwards, we are able to adjust reference targets by converting each size record in the mapping table to the corresponding offset in the *.newsec* section. Essentially, we add a level of indirection by replacing all indirect branch instructions with a direct branch to the mapping routine and consulting the mapping table for computing the new target address.

For instance, in Figure 2, the runtime value of indirect branch (*blx r2*) target address is 0x804816c. As shown in Figure 3, μSBS rewrites instructions in the *.newsec* section with base address 0x8200000. At runtime, the mapping routine looks for 0x804816c entry in mapping table in order to find the offset of new target address (0x81d0), and then returns new translated target address (0x8200000 + 0x81d0 = 0x82081d0). Finally, firmware jumps (*ldr pc, [sp, #-4]*) to the translated address.

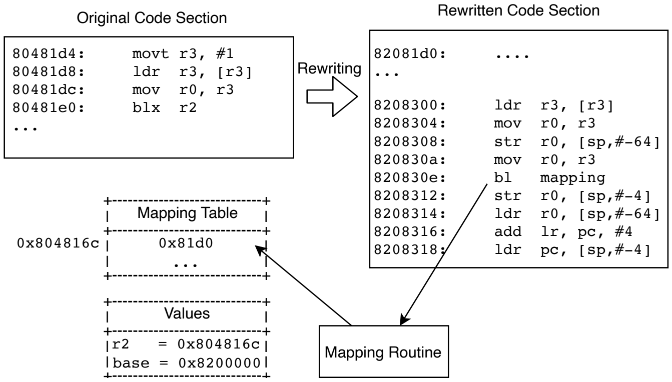
Algorithm 1: Generating a new code section

```
newsecGenerator ($Insts$)
    inputs : $Insts = inst_1 \ldots inst_n$
    output : $newsec$ section
    foreach disassemled instruction $inst_i \in Insts$ do
        if $inst_i$.type is a branch_instruction then
            if $inst_i$.ref is static then
                $inst_i.ref := AdjustTarget(inst_i.ref)$;
                $WriteInst(newsec, inst_i)$
            else
                $WriteInst(newsec, mapping\_instructions)$
        else if SanitizationSpecification($inst_i$) then
            if $inst_i$.type is a memory_allocation then
                $WriteInst(newsec, redzone)$;
                $WriteInst(newsec, inst_i)$;
                $WriteInst(newsec, redzone)$;
            else
                $WriteInst(newsec, metadata\_check)$;
                $WriteInst(newsec, inst_i)$;
        else
            $WriteInst(newsec, inst_i)$
    return $newsec$;
```

3.3 Sanitization Specification

The sanitization specification determines which exact instructions should be instrumented by μSBS. AddressSanitizer [46] and Valgrind's Memcheck [48] are the most widely adopted sanitizers for detecting memory safety violations in practice [52]. Inspired by these approaches, μSBS utilizes a metadata store that keeps the status of allocated memory bytes. μSBS surrounds every memory value with a so-called redzone representing out-of-bounds memory and marks it as invalid memory in the metadata store. Then, μSBS instruments every memory instruction (i.e., load and store) in order to consult the metadata store whenever the firmware attempts to access memory. Any access to a red-zone or to an unallocated memory region is considered as a memory corruption vulnerability and triggers a warning close to the location of the bug.

The current version of μSBS sanitizer enables us to observe faulty states caused by various types of spatial and temporal memory corruptions including: (1) Overrunning and underrunning heap blocks. (2) Overrunning the top of the stack. (3) Accessing memory after it has been freed. (4) Using memory values that have not been initialized or that have been derived



```
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]
```

```
0x81d0
...
```

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

Figure 3: Mapping procedure: all indirect branch instructions (e.g., blx r2) are firstly redirected to the mapping routine which looks for new offset (0x81d0) corresponding to the old target address (0x804816c) in mapping table.

from other uninitialized values. (5) Incorrect freeing of heap memory, such as double-freeing heap blocks.

4 Implementation

We have implemented a proof-of-concept of μSBS for the ARMv7-M architecture [1], which covers a large share of microcontrollers (i.e., Cortex-M3/4/7) for embedded platforms [35]. The following outlines the technical details of the implementation based on the design described in the previous section.

4.1 Binary Instrumentation

We implemented μSBS *binary instrumentation* component with a total of 1609 LOC in Python language using the Capstone framework [3] as our underlying linear disassembler engine. We used the pyelftools [8] and pwntools [7] open source frameworks for parsing the ELF files and reassembling instrumented assembly code respectively.

As mentioned in § 3.2, μSBS generates a new stretched code section (*.newsec*) and executes firmware from its new entry point. μSBS rewrites all instructions of the old code section together with new inserted instructions in *.newsec*. However, inserting new instructions may push a target address beyond the reach of the instruction referencing it. To resolve this issue, μSBS replaces every referencing instruction by its substitute which allows for larger offsets. For instance, it replaces *b* branch instructions with *b.w* to generate a 32-bit instead of 16-bit instruction. Furthermore, it is necessary to rewrite all branch instructions to handle the challenges mentioned in § 3.2.

To safely handle reference targets and adjust the binary layout, μSBS rewrites all direct branch instructions by statically changing their offsets. Thereafter, it rewrites all indirect branch instructions to deploy the dynamic mapping procedure. To do so, every indirect jump instruction with target address *label* must be rewritten by instructions shown in Listing 2.

```
str   r0, [sp, #-64]
mov   r0, label
bl    mapping
str   r0, [sp, #-4]
ldr   r0, [sp, #-64]
ldr   pc, [sp, #-4]
```

Listing 2: Rewritten instructions for every indirect jump instruction with target address label.

Similarly, every indirect call instruction with target address *func* must be rewritten by instructions shown in Listing 3. To store the return address, μSBS loads into Link Register (LR) the address of the instruction following the call to *[sp, #-4]*.

```
str   r0, [sp, #-64]
mov   r0, func
bl    mapping
str   r0, [sp, #-4]
ldr   r0, [sp, #-64]
add   lr, pc, #4
ldr   pc, [sp, #-4]
```

Listing 3: Rewritten instructions for every indirect call instruction with target address func.

μSBS translates every indirect branch into the *mov* and direct call instructions. The *mov* instruction puts the old target address into register *r0* and the direct call goes to the mapping routine which searches for the offset corresponding to the old target in the mapping table. If the search succeeds, it will jump to the translated target address in *.newsec*.

Finally, μSBS uses LIEF framework [9] to add *.newsec* and mapping routine to the original firmware ELF file. The LIEF framework modifies the ELF header and creates a new code segment containing the *.newsec* code section.

4.2 Sanitization

We implemented the process of generating a sanitization specification with 687 LOC in the Python language. Once the firmware binary is disassembled, μSBS extracts all object allocations and memory accesses (i.e., LDR and STR) and stores them in the sanitization specification file. The *binary instrumentation* component interprets the sanitization specification file for instrumenting all memory accesses with memory check instructions to consult the metadata store. The memory check instruction computes the address of the corresponding metadata byte, loads that byte and checks whether it is valid. For efficiency reasons, μSBS deploys a similar metadata management mechanism to AddressSanitizer by storing 1 byte of metadata for every 8 bytes of firmware memory. In this case, the metadata mapping accords with Formula 1 where meta_base is the base address of the metadata store

Table 2: Comparison of fault observability between μSBS and the heuristic-based method proposed by Muench et al. [42].

Memory Corruptions	Muench et al.	μSBS
Null Pointer Dereference	✓	✓
Stack-based buffer overflow	✓	✓
Heap-based buffer overflow	✓	✓
Format String	✓	✓
Double Free	✓	✓

and block_addr is the address of the memory block.

$$\boxed{meta\_addr = meta\_base + (block\_addr >> 3)} \qquad (1)$$

5 Evaluation

We evaluated μSBS from three different angles: (1) Whether it can make faulty states observable on binary firmware in an automatic fashion. (2) Whether it can rewrite the code section of binary firmware without breaking its functionality. (3) Assessing the runtime performance and size expansion of rewritten binaries in practice.

To that end, we investigated μSBS ability to catch the same class of memory bugs of the state-of-the-art [42] based on the WYCNINWYC vulnerable application [15, 42] (§ 5.1). We also conducted an evaluation on 11 real firmware images to check the correctness of the rewritten binaries by using the standard test suite that provided by the vendor of each firmware (§ 5.2). Finally, we measure the runtime performance of our rewritten and instrumented binaries (§ 5.3).

5.1 Effectiveness

We designed and performed this experiment to verify whether μSBS can successfully make the same class of memory corruptions observable compared to state-of-the-art fault observation method proposed by Muench et al. [42].

Experiment Setup: We used WYCNINWYC application, developed and used by Muench et al. as a testbench, in order to obtain comparable results. The WYCNINWYC application is a vulnerable implementation of an XML parser containing five different instances of spatial and temporal memory corruptions. Experiments are performed on a same development board, STM32-Nucleo L152RE [14] featuring an ARM Cortex-M3 CPU.

Experiment Results: We instrumented the WYCNINWYC application using the μSBS sanitizer and collected the statistics and results, including the observability of faulty states caused by memory corruptions. As shown in Table 2, μSBS caught all faulty states without the need for reverse engineering or advanced data-flow analysis techniques as required by the method of Muench et al. [42].

5.2 Feasibility

We performed this experiment to verify the correctness of the μSBS design and its application to large real-world bare-metal firmware. We rewrote the code sections of 11 binary firmware images without sanitization and observed whether all rewritten binaries executed correctly and produced identical output to the original.

Experiment Setup: We selected 11 real bare-metal binary firmware images for different applications, ranging from cameras to industrial control systems. These are full-fledged firmware and demonstrate the use of a diverse set of peripherals including an LCD Display, Microphone, Camera, Serial port, Ethernet and SD card. They collectively cover ARM Cortex-M3 and Cortex-M4 microcontrollers. In what follows, we provide a brief description of each firmware.

Audio-Playback firmware is developed for playing audio files by reading data from USB and sending it to the audio codec. *LCD-Display* is a firmware for reading and displaying a series of bitmaps from an SD card to the LCD. *LCD-Animate* displays animated pictures saved on a microSD card on the LCD. To create animated pictures, the firmware displays an images sequence with a determined frequency on the LCD. *Camera-USB* uses the Digital Camera Interface (DCMI) to connect with a camera module and display pictures on an LCD in continuous mode while also saving these pictures on the USB device. *FatFs-uSD* creates a FAT file system on the microSD and uses FatFs APIs to access the FAT volume in order to perform writing and reading of a text file. *TCP/UDP-Echo-Client/Server* are four firmware for running TCP/UDP echo client/server applications over Ethernet based on LwIP, a popular TCP/IP stack for embedded devices. *mbed-TLS* firmware runs an SSL client application based on the mbed-TLS crypto library and the LwIP TCP/IP stack for the STM32F4 family. PLC (Programmable Logic Controller) is a family of embedded devices for controlling critical processes in industrial environments. The *ST-PLC* firmware implements a PLC that executes uploaded ladder logic programs. The ladder logic program is uploaded to the microcontroller from an Android application via WiFi (ladder logic is a common PLC programming language).

All of these firmware images are provided with the development boards and written by STMicroelectronics [10]. Experiments are performed on STM32-Nucleo F401RE [11], STM32F479I-Eval [12], and STM32F4Discovery [13] development boards featuring an ARM Cortex-M4 CPU and STM32-Nucleo L152RE [14] featuring an ARM Cortex-M3 CPU.

Experiment Results: We executed both the rewritten version and the original version of the firmware on the test suit shipped with the firmware and compared their functional correctness. All of the rewritten firmware passed the functionality test and ran correctly, producing the same result as the original firmware.

Table 3: Statistical metrics of μSBS binary rewriting when applying to bare-metal firmware.

Firmware	MCU	Dir. Inst.	Ind. Inst.	Code (KB)	Rew. Code (KB)	Size Inc. (%)
Audio-Playback	STMF479I-Eval	1853	250	132	195	24
LCD_Display	STMF479I-Eval	809	103	48	57	10
LCD_Animate	STMF479I-Eval	803	103	48	56	10
FatFs_uSD	STMF4Discovery	575	150	23	29	6
TCP_Echo_Client	STMF479I-Eval	1407	132	79	91	14
TCP_Echo_Server	STMF479I-Eval	1384	132	77	89	14
UDP_Echo_Client	STMF479I-Eval	1341	132	76	88	14
UDP_Echo_Server	STMF479I-Eval	1310	130	75	87	14
Camera-USB	STMF479I-Eval	1003	163	70	81	13
mbed-TLS	STMF401RE Nucleo	3218	338	171	215	20
ST-PLC	STMF479I-Eval	2275	373	168	231	67

Table 3 presents the rewriting statistics and the modifications made by μSBS to the binary firmware images. The column under *Dir. Inst.* in the table represents the count of direct branch instructions, including calls and jumps, that are statically rewritten by changing their offsets. The column under *Ind. Inst.* represents the count of indirect branch instructions redirected to the mapping routine by μSBS.

Additionally, columns *Code* and *Rew. Code* represent the sizes of original code section and the rewritten code section (*.newsec*) respectively. The code section size overhead correlates positively with the number of indirect branch instructions due to the rewriting procedure that replaces each indirect branch with 6/7 instructions (§ 4.1). Furthermore, the fixed overhead of the mapping table and mapping routine play a significant role in the size overhead of the binary file (last column). In fact, the size overhead in percent will be less for large firmware compared to small images. For example, the original size of the *ST-PLC* firmware is 1.1MB and has 67% overhead, while *mbed-TLS* is 3.5MB and has only 20% overhead.

5.3 Performance

μSBS with no sanitization slows down firmware execution for two reasons. First, the firmware is statically instrumented and there is an additional direct call added for each indirect branch instruction to call the mapping routine. Second, μSBS dynamically searches for the new target address of every indirect branch instruction in the mapping table which incurs runtime overhead. In this section, we evaluate the execution overhead of rewritten binaries without sanitization. In addition, we also measure the execution overhead incurred by our sanitization procedure and the processing time of μSBS itself.

Each firmware in our benchmark was instrumented and executed twenty times. For example, we executed *LCD_Display* original and instrumented firmware for displaying 5 images twenty times. Figure 4 presents the runtime overhead results. The average slowdown for our benchmark without sanitization is 8.5%. The second bar in Figure 4 presents the execution overhead of instrumented firmware with the μSBS sanitizer. It increases the execution overhead to an average of 32.5% compared to rewritten firmware without sanitization. *Audio-Playback*, *mbed-TLS*, and *ST-PLC* are memory-intensive firmware, and therefore we expected the high overhead results arising from a large number of memory accesses that are expensive after being instrumented and checked. While, these overheads are not negligible, we believe that they are reasonable as this overhead is only incurred on the devices under security test and not the devices that are actually deployed on the field.

Figure 5 presents how long it takes μSBS to rewrite and sanitize firmware binaries. As expected, larger binaries take more time to be processed. On average, μSBS spends 5.72 seconds on binaries in our benchmark. We regard this as a promising result compared to state-of-the-art fault observation method [42]. The efficiency of μSBS makes it a practical tool for the large-scale sanitization of firmware binaries.

6 Related Work

This section provides an overview of the state-of-the-art. Related work can be categorised in works related to: (1) fault observation and (2) binary rewriting.

6.1 Fault Observation

Muench et al. [42] proposed the only existing method for fault observation in embedded systems by introducing six heuristics such as heap object tracking. They implemented these heuristics on top of a combination of the Avatar [61], PANDA [29], and Boofuzz [2] frameworks. However, these heuristics suffer from their reliance on a variety of information such as: memory accesses, memory mappings, executed instructions, register state and allocation and deallocation functions. This information must be extracted from target bi-

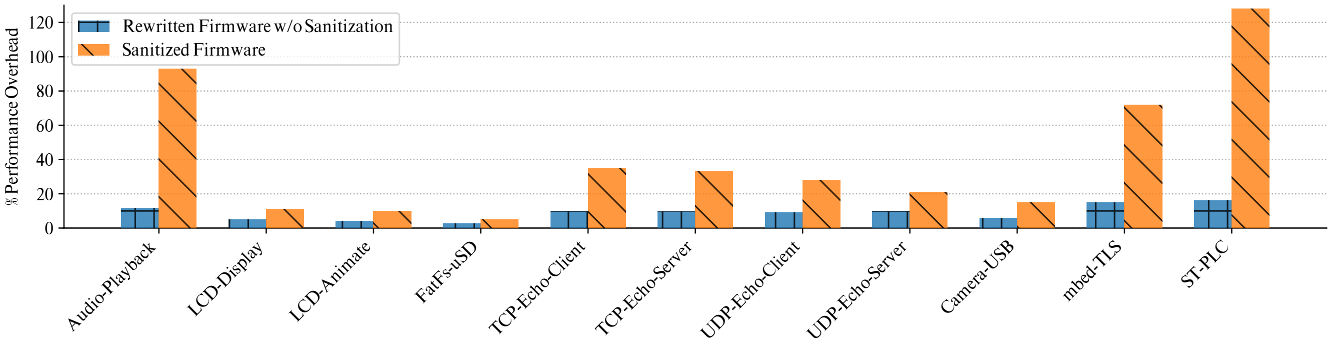


Figure 4: Performance impact of μSBS on 11 real firmware binaries with and without sanitization.

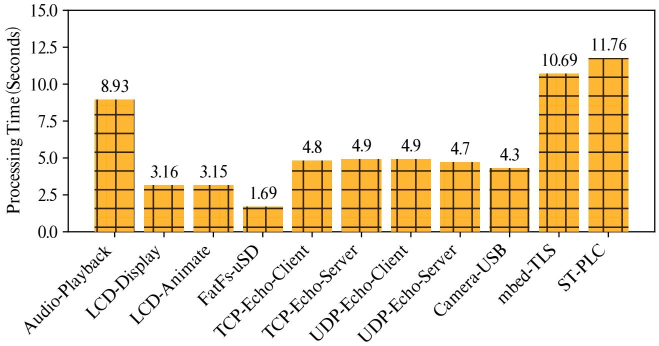


Figure 5: μSBS processing time for 11 real firmware binaries.

nary through reverse engineering and advanced static analysis, which adds both imprecision and complexity. Furthermore, applying heuristics for fault observation results in false positives and false negatives. Instead, μSBS approach turns out to be not only lightweight, avoiding heavyweight manual firmware analysis, but also reliable, being capable of providing platform-agnostic fault observation.

6.2 Binary Rewriting

Binary rewriting refers to the process of modifying one binary into another while one or more new instructions are optionally inserted to provide new features or behaviors. Binary rewriting methods can be categorized into two main classes: dynamic and static methods. Approaches [22,40,43] that belong to the first category transform stripped binaries that are loaded into memory while they are executing. However, they are not practical for fault observation on bare-metal devices due to the high performance overhead and special software/hardware requirements. HALucinator [26] is the state-of-the-art approach for dynamic binary instrumentation of bare-metal devices. In addition to the significant performance penalty, HALucinator only supports a small number of microcontrollers. HALucinator emulates firmware that use a Hardware Abstraction Layer (HAL), and it is only applicable when the HAL is available to the analyst. Consequently, HALucinator is not mature enough for instrumentation in large-scale stripped firmware binaries. BinRec [17] is a dynamic binary rewriter that leverages multiple dynamic analysis techniques to lift binaries into LLVM IR. BinRec is built on top of S2E [25] and QEMU virtual machine [21] that do not support bare-metal firmware.

There are a number of static binary rewriting approaches that transform binaries before execution. These approaches differ from each other in how they transform binaries without breaking their functionality and semantics. Solutions such as Bistro [27] and STIR [56] redirect control flow from the original location to trampoline code containing new instructions. Trampoline-based rewriters are able to preserve application semantics after instrumentation, at the cost of considerable performance and memory penalties.

Uroboros [55] presents a set of heuristics for recognizing references among integer values and converting them into assembler labels in order to generate a relocatable assembly code. Ramblr [54], built atop angr [49, 50], is a similar approach that improves Uroboros using a composition of static analyses and heuristics. Unfortunately, heuristics-based approaches suffer from false positives and negatives that result in broken reassembled binary. RetroWrite [28] and Egalito [59] provide an instrumentation method that uses relocation information which is only available in position independent codes. This is not a practical solution for firmware binaries that are statically linked. Multiverse [20] leverages a superset disassembling technique [38, 39, 58] and disassembles at each offset of the binary code to produce a superset of instructions. Multiverse binary rewriter is built on top of the disassembler to instrument all superset instructions. As noted by Miller et el. [41], superset disassembly has a substantial code size overhead (763% on SPECint 2006 benchmarks). Furthermore, experimental results [59] show that Multiverse does not support statically linked binaries.

All the above static approaches are designed and developed for the x86 architecture. RevARM [36] is the only static binary rewriter proposed for instrumenting ARM-based mo-

Table 4: A subjective comparison between μSBS and state-of-the-art binary rewriting methods proposed for General Purpose (GP) computers, Mobile (M), and Bare-metal (B) devices. X and A denote x86 and ARM architectures respectively.

Objectives	Uroboros [55]	Ramblr [54]	RetroWrite [28]	Egalito [59]	Multiverse [20]	RevARM [36]	μSBS
Target	GP	GP	GP	GP	GP	M	B
Architecture	X	X	X	X	X	A	A
w/o Heuristics	✗	✗	✓	✓	✓	✗	✓
w/o Relocation	✓	✓	✗	✗	✓	✓	✓
w/o IR Lifting	✓	✓	✓	✓	✓	✗	✓

bile applications. RevARM lifts binary code to a higher-level intermediate representation (IR) and performs the instrumentation procedure at that level. As pointed out by Dinesh et al. [28] lifting a binary to an IR usually misses application semantics and actual control flows since it is necessary to precisely model instruction set architecture (ISA) and extract type information from the binary file. Additionally, RevARM uses the Uroboros technique for differentiating references and integer values which is an impractical solution that is unable to work on non-trivial application binaries. Table 4 presents a subjective comparison of state-of-the-art binary rewriting approaches and μSBS.

7 Discussion

In this section, we discuss the limitations in our system and shed some light for future work.

Supported microcontrollers. This paper focuses on a specific subclass of embedded microcontrollers running a single statically linked firmware—bare-metal firmware. Like all other static binary instrumentation methods, we do not handle any dynamically loaded code. Support for sanitizing such code requires dynamic instrumentation since such code can only be seen while the firmware is running.

Supported CPU architectures. The current implementation of μSBS supports ARMv7-M architecture as the most widely used core for embedded systems [35]. Our platform-independent approach can support bare-metal firmware developed for other architectures like x86 with a small extra engineering effort since they are comparable or have more relaxed requirements for the binary instrumentation purpose [36]. For example, it is mainly required to change the assembly language of the rewritten and inserted instructions and mapping function in order to support x86 firmware.

Fuzzing. Although the current μSBS implementation has focused on the observation of faulty states due to the memory corruptions, it may be extended and integrated with fuzzing methods to uncover new bugs in bare-metal firmware. To be concrete, μSBS sanitizer can be leveraged to improve the bug-finding ability of fuzzing methods [47,63]. More specifically, we may guide the input generation process of the fuzzers towards triggering μSBS sanitizer checks. Improvement on IoT fuzzing [31,32,62] is orthogonal to this paper, and we will leave it for future work.

Sanitization. μSBS can potentially observe a wide array of memory corruptions by applying memory safety policies. The current implementation of μSBS sanitization process is inspired by AddressSanitizer and Valgrind's Memcheck policies. However, other sanitization techniques can be developed on top of μSBS *binary instrumentation* component for observing faulty states caused by other types of memory corruption vulnerabilities. We leave such improvements to future work.

8 Conclusion

Memory corruption vulnerabilities are common in IoT firmware binaries and can lead to significant damage on bare-metal embedded devices that are increasingly intertwined with critical industrial and medical processes. In this paper, we have presented a concrete investigation of hardware security feature (i.e., MMU, MPU, and DEP) in a representative selection of IoT SoC families. Our analysis shows that the IoT fuzzing world lags behind the general-purpose world. We have also developed and demonstrated μSBS, the first fully automatic approach for observing faulty states in bare-metal firmware. μSBS uses a novel combination of static binary instrumentation and sanitization to validate memory accesses in a firmware binary, allowing for an improved fault observation mechanism. We evaluated μSBS using a fault observation benchmark and 11 real firmware binaries. Our approach correctly sanitized all the firmware binaries with reasonable run-time over-head and size expansion while discovering the same set of vulnerabilities as the state-of-the-art. To motivate further research in this field and encourage reproducibility, we open-source μSBS at https://github.com/pwnforce/uSBS.

Acknowledgments

This research is supported by the research fund of KU Leuven and imec, a research institute founded by the Flemish government. The work of the third author has been partially supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe. We are grateful to anonymous reviewers for assisting us with their helpful comments and criticisms.

References

[1] ARMv7-M architecture reference manual. https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf. Accessed: February 2020.

[2] BooFuzz source code repository. https://github.com/jtpereyda/boofuzz. Accessed: February 2020.

[3] Capstone: The ultimate disassembler framework. http://www.capstone-engine.org/. Accessed: February 2020.

[4] CVE-2017-0561. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0561. Accessed: February 2020.

[5] Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016. Accessed: February 2020.

[6] Google Project Zero: Over The Air: Exploiting Broadcoms Wi-Fi Stack. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html. Accessed: February 2020.

[7] Pwntools: CTF framework and exploit development library. https://github.com/Gallopsled/pwntools/. Accessed: February 2020.

[8] Pyelftools: Parsing ELF and DWARF in Python. https://github.com/eliben/pyelftools/. Accessed: February 2020.

[9] Quarkslab Lief project. https://lief.quarkslab.com/. Accessed: February 2020.

[10] STM32Cube MCU Packages. https://www.st.com/en/embedded-software/stm32cube-mcu-packages.html. Accessed: February 2020.

[11] STM32F401RE MCU. https://www.st.com/en/evaluation-tools/nucleo-f401re.html. Accessed: February 2020.

[12] STM32F479I MCU. https://www.st.com/en/microcontrollers-microprocessors/stm32f469-479.html. Accessed: February 2020.

[13] STM32F4DISCOVERY MCU. https://www.st.com/en/evaluation-tools/stm32f4discovery.html. Accessed: February 2020.

[14] STM32L152RE MCU. https://www.st.com/en/evaluation-tools/nucleo-l152re.html. Accessed: February 2020.

[15] WYCNINWYC application source code repository. https://github.com/avatartwo/ndss18_wycinwyc. Accessed: February 2020.

[16] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46. IEEE, 2019.

[17] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. BinRec: Dynamic binary lifting and recompilation. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2020.

[18] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600. USENIX Association, 2016.

[19] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860. USENIX Association, 2014.

[20] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[21] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2005.

[22] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 265–275. IEEE, 2003.

[23] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 213–223, 2011.

[24] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[25] Vitaly Chipounov, Volodymyr Kuznetsov, , and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[26] Abraham 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 *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2020.

[27] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: binary component extraction and embedding for software security applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2013.

[28] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P')*, 2020.

[29] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015.

[30] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[31] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[32] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated rehosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150. USENIX Association, 2019.

[33] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[34] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the winter 1992 USENIX conference*. USENIX Association, 1991.

[35] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[36] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. RevARM: A platform-agnostic ARM binary rewriter for security applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pages 412–424. ACM, 2017.

[37] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 423–427. Springer, 2008.

[38] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2004.

[39] Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and George Portokalidis. GPU-Disasm: A gpu-based x86 disassembler. In *Proceedings of the international Information Security Conference*, pages 472—489, 2015.

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, 2005.

[41] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198. IEEE, 2019.

[42] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[43] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, 2007.

[44] Majid Salehi, Danny Hughes, and Bruno Crispo. Microguard: Securing bare-metal microcontrollers against code-reuse attacks. In *Proceedings of the IEEE Conference on Dependable and Secure Computing (DSC)*, 2019.

[45] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, 2002.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 309–318, 2012.

[47] Kostya Serebryany. Sanitize, fuzz, and harden your c++ code. In *USENIX Enigma*. USENIX Association, 2016.

[48] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2005.

[49] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Network and Distributed System Security Sym. (NDSS)*, 2015.

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 138–157. IEEE, 2016.

[51] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1275–1295, 2019.

[52] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 48–62. IEEE, 2013.

[53] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable use-after-free detection. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[54] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[55] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642. USENIX Association, 2015.

[56] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[57] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 299–308. ACM, 2012.

[58] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Proceedings of the Pacific-Asia Conference Knowledge Discovery and Data Mining (PAKDD)*, pages 273—-285, 2014.

[59] David Williams-King, Hidenori Kobayashi, Kent Williams-King, G. Elaine Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[60] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2010.

[61] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[62] 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 *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114. USENIX Association, 2019.

[63] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.