Facilitating Non-Intrusive In-Vivo Firmware Testing with Stateless Instrumentation

Jiameng Shi
University of Georgia
jiameng@uga.edu

Wenqiang Li
Independent Researcher
wenqiang-li@outlook.com

Wenwen Wang
University of Georgia
wenwen@cs.uga.edu

Le Guan
University of Georgia
leguan@uga.edu

***Abstract*—Although numerous dynamic testing techniques have been developed, they can hardly be directly applied to firmware of deeply embedded (e.g., microcontroller-based) devices due to the tremendously different runtime environment and restricted resources on these devices. This work tackles these challenges by leveraging the unique position of microcontroller devices during firmware development. That is, firmware developers have to rely on a powerful engineering workstation that connects to the target device to program and debug code. Therefore, we develop a decoupled firmware testing framework named *IPEA*, which shifts the overhead of resource-intensive analysis tasks from the microcontroller to the workstation. Only lightweight "needle probes" are left in the firmware to collect internal execution information without processing it. We also instantiated this framework with a sanitizer based on pointer capability (*IPEA-San*) and a greybox fuzzer (*IPEA-Fuzz*). By comparing *IPEA-San* with a port of AddressSanitizer for microcontrollers, we show that *IPEA-San* reduces memory overhead by 62.75% in real-world firmware with better detection accuracy. Combining *IPEA-Fuzz* with *IPEA-San*, we found 7 zero-day bugs in popular IoT libraries (3) and peripheral driver code (4).**

I. Introduction

Microcontroller units (MCUs) are resource-constrained *System-on-Chip* (SoCs) that have found wide applications in *Internet-of-Things* (IoT), industrial control, smart manufacturing, healthcare, etc. The secure operation of these systems therefore heavily relies on the security of firmware running on the MCU-based devices. In recent years, we have witnessed a significant number of vulnerability exposures that target MCU-based systems [45], [53], [46], [22], [36], [37], [71], resulting in widespread real-world exploitation [30], [40], [48], [84]. This highlights the critical need to develop effective and efficient firmware testing tools. Unfortunately, other than rudimentary debugging tools (e.g., halt-and-examine style debugging), firmware developers rarely enjoy modern (dynamic) software testing techniques (e.g., sanitizers).

The slow uptake of established dynamic analysis techniques for firmware can be explained by the significant disparity in the development and testing environment. Specifically, firmware developers write the source code on a development workstation/PC, where the code is cross-compiled into binary-format firmware. To transfer the firmware to the target hardware (i.e., prototype board) and debug its execution, a debug dongle that connects the board to the development PC is used. Notably, firmware development and firmware execution take place on different machines (development PC vs. MCU prototype board). This poses a dilemma regarding where to run firmware analysis. On the one hand, the target MCU, with very restricted resources, may not have spare memory and computation power to independently host a test. On the other hand, the development PC, without the facility to access the internal state of firmware execution from outside, cannot gather insightful data to conduct effective analyses.

To approach this dilemma, this work presents a new firmware testing framework named *IPEA* (short for *in-vivo probe, ex-vivo analysis*). Instead of running a test entirely on the development PC or the board, our key innovation is to partition the workload of firmware testing into two *decoupled* parts. A small part is kept on the MCU chip to collect necessary run-time information with high fidelity (in-vivo aspect). The remaining part, which conducts the actual resource-hungry analysis is offloaded to the more capable development PC (ex-vivo aspect). The communication between the two—transmitting the collected information from the MCU to the PC—is facilitated by the debug dongle, a must-have component in the existing firmware development environment.

Since the two decoupled parts on the PC and MCU board must jointly complete a test, an immediate challenge is to minimize the frequency of interactions between them, as the round-trip time is often not negligible. We address this problem with a new notion named *stateless instrumentation*. With it, the firmware is instrumented such that it never stores or processes metadata related to the analysis. Rather, it only places "needle probes" to collect and stream out important events happened during execution. The ex-vivo analysis task on PC then reconstructs the run-time metadata based on the received information and performs the actual analysis. Stateless instrumentation offers two benefits. First, when the firmware is executing, only one-way communication is needed, avoiding the interaction between the PC and MCU that may block the normal execution. Second, since the MCU does not need to store any metadata, the memory overhead can be significantly reduced.

The *IPEA* framework features three user-friendly properties, with which we expect to ultimately promote the adoption of advanced dynamic analysis techniques in the MCU community. **Non-intrusive:** *IPEA* seamlessly integrates into existing firmware development workflow and does not rely on additional hardware. **In-vivo:** The essential run-time in-

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23116
www.ndss-symposium.org

formation for firmware analysis is collected directly on the target MCU, instead of from an external observer (e.g., [36], [32], [91]). **Lightweight:** The testing incurs low overhead on resource-constrained MCUs, especially in terms of memory consumption.

While not all software testing techniques can take full advantage of *IPEA* (e.g., when they cannot be easily decoupled), our research indicates that many bug-oriented testing techniques such as fuzzing and various sanitizers can benefit a lot from it. As such, we further design and implement two plugin modules for *IPEA* to demonstrate its real-world application: a fuzzer (*IPEA-Fuzz*) and an address sanitizer (*IPEA-San*). *IPEA-Fuzz* implements a greybox fuzzer with edge coverage as feedback, similar to AFL [88]. *IPEA-San* is a pointer-based sanitizer that tracks the bounds and validity of each pointer with intra-object overflow detection support.

Our prototype, including the framework and two analysis plugin modules, have been extensively tested on Arm Cortex-M series MCUs. It is worth pointing out that our approach is not limited to a specific MCU architecture, as it does not rely on any hardware features. To compare *IPEA-San* with state-of-the-art solutions that run entirely on the test target, we ported Google's AddressSanitizer (or ASan) [70] for MCU, which is memory-optimized to accommodate MCU-specific restrictions. Even so, *IPEA-San* exhibits significant advantages over ASan. *IPEA-San* incurs zero false negatives and false positives on the Juliet Test Suite and consumes far less RAM in real-world firmware (1.14x vs. 3.06x). Constrained by the RAM capacity, two demonstration projects that come with the official commercial off-the-shelf (COTS) development kits failed to compile using ASan but succeeded using *IPEA-San*. When running *IPEA-San* against the BEEBS benchmark [61], *IPEA-San* reported two "silent" memory corruptions caused by bugs that are *not* known before. Last but not least, combining *IPEA-Fuzz* with *IPEA-San*, we found seven zero-day bugs, including three in popular IoT libraries and four in peripheral driver code.

In summary, we made the following contributions.

- **New framework –** We propose *IPEA*, a decoupled framework that enables non-intrusive, in-vivo and lightweight firmware testing.
- **New analysis techniques –** We design and implement a sanitizer and a fuzzer as plugins for the *IPEA* framework.
- **New porting for MCUs –** To comprehensively evaluate *IPEA-San*, we adapt the Juliet Test Suite for MCUs. To compare *IPEA-San* with the state of the art, we complete a memory-optimized port of ASan for MCUs.
- **Evaluation –** We evaluate *IPEA-San* against the Juliet Test Suite, the BEEBS benchmark, and 12 real-world firmware samples. *IPEA-San* reduces RAM overhead by 62.75% compared with ASan and incurs zero false negatives and false positives in the Juliet Test Suite.
- **New bugs –** *IPEA-San* exposed two memory bugs in the BEEBS benchmark. Combining *IPEA-Fuzz* with *IPEA-San*, we found seven zero-day bugs in real MCU products.

All the code, benchmark suites, and firmware samples developed in this work have been open-sourced to encourage continued research on GitHub (https://github.com/MCUSec/IPEA) and Zenodo [74].

II. Background

A. Firmware Development

In a typical firmware development environment, developers write source code on a development PC and cross-compile it into the firmware. A hardware debug dongle acts as a bridge between the two and enables firmware downloading and debugging. It has the ultimate control over the target MCU, including suspending/resuming firmware execution, placing a breakpoint, examining the register/memory values, etc. These control commands are generated by the debug daemon (e.g., GDBSever) on the PC. The dongle then translates them into low-level JTAG/SWD messages understandable by MCUs.

B. Microcontroller Firmware

Compared with traditional microprocessors found on PCs and smartphones, MCUs consume less power, run at lower frequencies (less than 500 MHz), and integrate on-chip SRAM and flash memory which are typically several hundred KB. The software on them, aka firmware, is tightly coupled with the underlying hardware. Due to the restricted resources, MCU devices commonly lack standard features on PCs (e.g., MMU). Notably, MCU firmware runs in a single flat address space where different components are mixed together, including the kernel (if any), user tasks and memory-mapped peripherals. This leads to a tremendously different runtime environment (bare-metal vs. Linux/Windows-based), making existing dynamic software testing tools inapplicable to MCU firmware. Specifically, dynamic analysis highly depends on the runtime environment of the target software. Without radical redesign to accommodate the changes, a tool oftentimes cannot be reused to analyze MCU firmware. Even if an existing tool can, the scarce resources on MCUs may make it too expensive.

C. Memory Bugs and Bug-Oriented Program Analysis

Memory Safety. To fully exploit the limited resources, MCU firmware is mainly written in the C/C++ programming languages. Unfortunately, these languages are memory-unsafe, which means attackers can potentially leverage implementation bugs to access memory not allowed by the original semantics (i.e., memory corruption). A spatial memory bug concerns out-of-bounds (OOB) accesses and a temporal memory bug concerns accessing a de-allocated object or an unintended object via a dangling pointer (e.g., use-after-free or UAF).

Fuzzer and Sanitizer. Combining a sanitizer and a fuzzer has been commonly used to reveal real-world memory bugs. A fuzzer generates abnormal testcases for the software-under-test in an attempt to trigger a memory safety violation. Although some memory corruptions can be immediately caught by the OS, others may not. Therefore, a sanitizer comes to the rescue and makes memory corruptions more noticeable. There are many types of memory errors that can be captured by different sanitizers [80], [81], [38], [82], [70]. Among them, address sanitizer is particularly powerful. The general idea is to instrument the target software to enforce bounds (spatial property) and/or validity (temporal property) checks at pointer dereferences. Any violation will generate an immediate alert instead of waiting for the OS to catch it. This role of sanitizer is particularly important for fuzzing embedded system firmware

since memory corruptions on these devices rarely lead to observable crashes due to the lack of efficient hardware-based memory isolation mechanisms [55].

Redzone-Based Sanitizer vs. Pointer-Based Sanitizer. Many address sanitizers have been proposed [70], [79], [33], [72], [56], [57], [58], [51], [90], [42], [20]. Two designs receive wide adoption: redzone-based solutions such as ASan [70] and pointer-based solutions such as CCured [58]. The former places redzones around objects. A shadow memory is maintained to bookkeep the state of memory and an alert is raised on accessing the redzone. To detect temporal memory bugs, it quarantines freed objects in a buffer. An alert is raised on accessing an object in the quarantine. The latter encodes the capability of each pointer via metadata, including its bounds and validity. Before each pointer dereference, the target address is checked against the per-pointer metadata. While earlier efforts use a fat pointer representation to replace a pointer with a multi-word pointer/metadata [58], recent advances tend to use disjoint metadata [56] or rely on hardware features [79] to improve compatibility.

Memory Tagging. Memory tagging is a variant of pointer-based sanitizer. It is based on the lock-and-key mechanism. When an object is allocated, its memory and receiving pointer are given the same tag. Then, all accesses to that memory must be made by a pointer having the same tag. Memory tagging simultaneously enforces the spatial and temporal properties because both OOB and UAF accesses lead to a mismatched tag. Memory tagging can be efficiently implemented in hardware. For example, Arm MTE (memory tagging extension) [17], an extension to Arm's Armv8.5-A architecture, reserves the upper four bits of a pointer to store the tag and the hardware transparently maintains tags for memory objects on 16-byte granules. At run-time, the CPU checks if the tag of the pointer and the tag associated with the target memory match on each load and store. Note that MTE is only available on Arm's A-profile processors, not MCUs.

III. Overview

Problem. Many software testing tools are developed for the emulated execution environment where the target code is translated on a host machine (e.g., Memcheck [72]). By dynamically instrumenting the translation, these solutions can transparently collect rich execution information with excellent scalability. As such, rehosting firmware on a PC seems to be a promising solution for firmware analysis. However, unlike traditional software, a prerequisite of firmware emulation is a precise model of the underlying hardware because firmware frequently interacts with diverse peripherals. Since peripheral behaviors are hard to predict, emulation often lacks the required fidelity and even leads to false crashes [31], [85].

Involving real hardware in the loop avoids challenges in emulating firmware. However, it comes with its own problems. First, running a test entirely on the target MCU device is restricted by the short of on-chip resources. Take ASan as an example. Due to the excessive use of redzones and shadow memory, the memory overhead of ASan is 3.37x on average [70]. One may argue that the testing overhead can be largely tolerated since it does not carry over to real products. However, we found that firmware—if bloated with instrumentation—frequently exceeds the capacity of the prototype board. The root cause is that firmware does not enjoy the flexibility of being tested on more powerful or even virtualized hardware as traditional software does. In fact, to maintain compatibility and reduce costs, manufacturers prefer to use a prototype board that has identical hardware specification as the real product, which merely integrates enough resources for implementing the designed application logic.

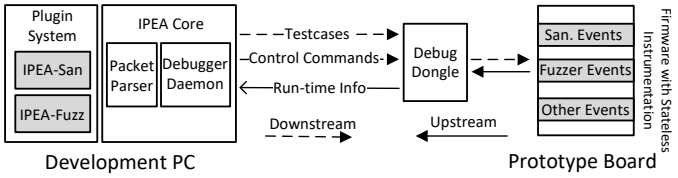

Fig. 1. *IPEA* overview.

Second, if we run the test on the development PC, the internal run-time state of firmware remains opaque, impeding effective firmware analysis. Prior endeavors either observe coarse-grained feedback of the execution (e.g., SweynTooth [36] obverses device responses to find BLE non-compliance bugs), or depend on special hardware to collect certain types of feedback (e.g., μAFL [50] leverages the ETM debug feature to collect execution trace). A general mechanism to collect arbitrary internal run-time information is missing.

Design Goals. Recognizing the barriers to effective firmware analysis, we design our system with the following goals. **G1**: It should not limit its application to certain types of analyses by relying on chip-specific features (as opposed to μAFL [50]). **G2**: It should run the firmware on real devices to collect high-fidelity information, instead of on emulators (as opposed to Memcheck [72]). **G3**: The memory overhead should be minimal so that the tool can be used on existing prototype boards (as opposed to ASan [70]).

Methodology. The key innovation of our approach is to strategically decouple the analysis workload and then selectively distribute the two parts on both the MCU and the development PC. The enabling technique is *stateless instrumentation*. With it, the MCU only runs the lightweight part of the test to collect high-fidelity execution information in-vivo, whereas the resource-hungry part of the test is *offloaded* to a powerful development PC. Since the development PC is a standard component in any MCU development environment, our solution does not impose additional deployment costs. Using static instrumentation via compiler techniques, our system meets **G1** and **G2**. With stateless instrumentation, **G3** is met.

System Overview. *IPEA* is a firmware testing framework. It does not include any analysis components in itself. Rather, developers load an analysis plugin into the plugin system on the PC and compile the target firmware using the analysis-specific LLVM pass, as shown in Figure 1. Here, the plugins, running on the PC, handle resource-hungry analyses offloaded from the target MCU, and the firmware runs stateless instrumentation that collects the analysis-specific execution information. The *IPEA* core on the development PC and the debug dongle together govern the communication between the analysis plugins and the target prototype board via three communication channels. The upstream **execution information channel** carries

the collected execution information from the board to the debugger daemon. Through the packet parser, the information is eventually routed to the intended plugins. During a test, other than upstreaming data via this channel, there is no interaction between the PC and MCU. This ensures that the firmware execution can never block. The two downstream channels are used by the *IPEA* core to prepare a test. In particular, the **control command channel** coordinates firmware execution and the **testcase channel** transfers testcases to the board.

We design two analysis plugins for the *IPEA* framework. *IPEA-Fuzz* implements a greybox fuzzer with edge coverage as feedback. *IPEA-San* is a pointer-based sanitizer. It virtually extends the capability of MCUs by emulating an enhanced version of Arm MTE hardware extension, in which the MCU and the PC plugin jointly complete the sanitization. The reason why we choose these two analyses is that combining a fuzzer and a sanitizer has been proven effective in finding real-world bugs. In what follows, we present the details of the *IPEA* framework, *IPEA-San* and *IPEA-Fuzz*, respectively.

IV. The *IPEA* Framework

System Setup. *IPEA* is designed for firmware developers. Therefore, we assume the availability of the source code and a typical embedded system development environment. In particular, a debug dongle and a debug daemon are present to bridge the target MCU and the development PC via the JTAG or Arm SWD interface. The dongle has ultimate control over the target MCU, including suspending/resuming the execution, placing a breakpoint, examining the register/memory values, etc. These commands are generated by the debug daemon (e.g., GDBSever) on the PC, which is then translated into low-level JTAG/SWD messages via the dongle.

***IPEA* Core.** The *IPEA* core runs on the PC and leverages the debug dongle to prepare the testing environment for each testcase to run on the MCU target. Its main tasks include 1) programming instrumented firmware to the MCU flash; 2) starting firmware execution; 3) examining the status of firmware execution as needed; 4) recovering the non-responsive hardware as needed; 5) downloading testcases to the target; and 6) receiving run-time data from the target. Among them, tasks 1-4 are standard functions already provided by most debug dongles. Tasks 5 and 6 are *IPEA*-specific, for which multiple communication interfaces can be used. For example, SEGGER Real Time Transfer (RTT) [68] and semihosting [18] provide chip-agnostic solutions, while chip-specific methods such as UART and Ethernet are also possible. In this work, we choose RTT as the underlying interface. This is because RTT is backed by the market-leading SEGGER J-Link solution [66]. Using it, our implementation can automatically support all MCUs based on Arm, RISC-V and Renesas RX [68]. In essence, RTT reserves a SEGGER RTT Control Block structure in the target MCU's memory to exchange data with the PC in the background with help of the debug dongle.

The *IPEA* core also needs to route the received packets to the intended analysis plugins. To facilitate this, we design a compact encoding scheme based on tag-value. For each type of run-time event to be collected, there is a unique header containing the identification tag followed by the value. The comprehensive list of supported events in this work can be found in Table I. The firmware is instrumented such that when an interesting event is captured, a function is inserted to stream out the encoded packet via RTT. Correspondingly, after receiving the packet, the packet parser checks the packet header and routes it to the target analysis plugin.

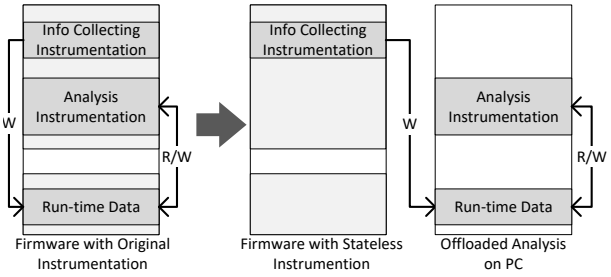

Fig. 2. Stateless instrumentation.

TABLE I. Run-time Events for Stateless Instrumentation

Event Opcode	Description
OP_NEW	Create a new heap object
OP_SUB	Derive a new pointer to a sub-object from an existing object
OP_FREE	Deallocate a heap object
OP_PROP	Intra-procedure tag propagation
OP_PROP_CALL	Inter-procedure tag propagation via parameters at callsite
OP_PROP_CALLEE	Inter-procedure tag propagation via parameters at callee
OP_PROP_RET	Inter-procedure tag propagation via return value at callee
OP_PROP_RET_GET	Inter-procedure tag propagation via return value after callsite
OP_CHK	Check invalid memory accesses
OP_CALLEE	Function call
OP_RET	Function return
OP_IRQ	Interrupt handler entry
OP_BB	Basic block random number

Stateless Instrumentation. *IPEA* offloads the resource-hungry firmware testing workload from the MCU to the development PC, which has "unlimited" resources compared with any MCU. To make this happen, we need to place some lightweight "needle probes" into the firmware so that they can collect necessary internal information, a notion we termed as *stateless instrumentation*. Figure 2 conceptually illustrates the idea. When we run firmware testing entirely on MCU, the original instrumentation can be separated into two parts. The information-collecting instrumentation collects the firmware run-time data, which is stored in a dedicated memory region. The analysis instrumentation performs the actual analysis using the collected run-time data. All three components (two parts of instrumentation plus the run-time data) reside in the address space of the firmware, as shown on the left of the figure. If the instrumentation can be refactored into stateless instrumentation, only the information-collecting part needs to be kept in the firmware and other parts can be offloaded, as shown on the right of the figure. However, not all kinds of testing can be easily refactored into stateless instrumentation, especially when the two parts of instrumentation are entangled with each other. We informally define two characteristics that a testing should have to benefit from stateless instrumentation. Condition i): The information-collecting instrumentation only writes to the run-time data. Condition ii): The analysis instrumentation and the run-time data are self-contained. Besides fuzzing and sanitization, we discuss how other security analyses can be supported in the *IPEA* framework in §IX.

V. *IPEA-San*: A Sanitizer for MCU Firmware

We first explain the choices we face in designing *IPEA-San* and the rationale behind our final decision. Then we use an intuitive example to demonstrate the basic idea, followed by detailed descriptions of how we instrument the firmware as well as how the PC plugin handles the data streamed from firmware. Finally, we discuss how to support third-party libraries in *IPEA-San* and optimization opportunities.

A. Design Choices

We found that both the redzone-based sanitizer and pointer-based sanitizer (see §II-C) are compatible with stateless instrumentation. However, pointer-based solutions present four distinct advantages over redzone-based solutions, compelling us to opt for designing our system based on pointer capability. First, the redzones and quarantine in redzone-based solutions by nature have to reside in the address space of firmware. Such overhead can hardly be offloaded. Second, redzone-based solutions may incur false negatives when a non-linear OOB access lands in a non-redzone location or the quarantine is exhausted, which is avoided in pointer-based solutions. Third, redzone-based solutions cannot support intra-object bounds checking since shadow memory cannot distinguish a sub-object from other fields. This can be solved in pointer-based solutions by maintaining distinct metadata for each pointer. Finally, besides bounds and validity, per-pointer metadata can be easily extended to track other information. For example, when type information is tracked, type safety can be checked (see §IX).

B. Basic Idea

IPEA-San is designed around memory tagging, a variant of pointer-based solutions. It virtually extends the capability of MCUs by emulating an enhanced Arm MTE hardware feature with intra-object overflow detection support. Under the *IPEA* framework, the MCU and the PC plugin jointly complete the sanitization. Specifically, the MCU does not need to maintain any metadata. Rather, it only collects and streams out information about object creation/deallocation, pointer propagation and pointer dereferences. The PC plugin can then emulate MTE by reconstructing the run-time metadata based on the received information and performing sanitization. It is worth noting that since *IPEA-San* involved a powerful PC to emulate MTE, it can overcome the limitations of the hardware-based MTE. For example, by simulating an unlimited number of tags, *IPEA-San* eliminates conflicts (thus false negatives) that may occur in systems based on hardware MTE [79]. Using the tag overlay technique (see §V-C1), *IPEA-San* also supports intra-object overflow detection. Current MTE implementation cannot support it since the hardware can only maintain one tag for a particular byte. To demonstrate the basic idea of *IPEA-San*, we use an intuitive example below, where the red dotted lines indicate the dependency between the instrumentation (highlighted in grey) and the source code.

```
1 p1 = malloc(size);
2 //collect bounds info for heap objects
3 send_to_PC(OP_NEW, p1, p1 + size, p1_id);
4 p2 = p1;          //p2 is a local variable
5 //propagate to temporary pointers
6 send_to_PC(OP_PROP, p2_id, p1_id);
7 g_p = p2;         //g_p is a global variable
8 //propagate to in-memory pointers
9 send_to_PC(OP_PROP, &g_p, p2_id);
10 ...
11 //check usage before dereferences
12 send_to_PC(OP_CHK, &(p2->a), sizeof(p2->a), p2_id);
13 value = p2->a;
```

Listing 1. The basic idea of *IPEA-San*.

This example only contains four statements, which allocate a new object on the heap, propagate the pointer (`p1`) to another local pointer (`p2`) and a global pointer (`g_p`), and access the field `a` of the object by dereferencing `p2`. To detect whether there is a buffer overflow, we insert four instrumentation functions. When a new object is allocated, line 3 sends out its bounds information along with the ID of the temporary receiving pointer. Here, `OP_NEW` indicates that this is a `malloc` operation. `p1` and `p1+size` represent the base and limit of the object respectively. To uniquely identify a pointer, an ID `p1_id` is used. After receiving such information, the sanitizer plugin on the PC assigns a unique tag to both that memory range and the temporary pointer. Pointer tags can propagate to another pointer, as instrumented at lines 6/9, meaning that the target pointers with ID `p2_id` and `&g_p` will have the same tag. We elaborate on the difference between temporary pointers and in-memory pointers in §V-C1. Finally, before dereferencing the object, the target address range and the ID of the dereferenced pointer are sent out, as shown at line 12. On PC, the tag of the target memory and the tag associated with the pointer will be compared.

C. IPEA-San Instrumentation

To detect memory safety violations, information about four kinds of events is essential: a) object creation, b) pointer propagation, c) pointer dereference, and d) object deallocation. For each of them, we first discuss how they are collected during firmware execution. Then we explain how they are used to reconstruct the metadata to facilitate sanitization on the PC.

1) Object Creation: Depending on the location of the object, we use different instrumentation strategies.

Heap Objects. We insert a function `send_to_PC(OP_NEW, base, limit, id)` after every allocation operation (e.g., `malloc`). The meaning of parameters has been explained before, but the `id` parameter which uniquely identifies the receiving pointer needs further clarification. When the address of allocated object is returned to an in-memory pointer (i.e., pointers that must be stored to and retrieved from memory), we use the address that stores this pointer as the `id`. This is feasible because this address is unique in the address space of firmware. However, when the address is returned to a temporary pointer (i.e., pointers held in local variables that can be promoted to registers), there is no address to use. To address this problem, a compile-time static ID is assigned for each of these receiving pointers (see the example in Listing 1). This design works fine if there is no reentrant code; otherwise, it cannot distinguish pointers from the same allocation site in different contexts (e.g., recursive invocations). A straightforward solution is to maintain a call stack for each execution thread on PC. To this end, in each function prologue, we also insert a function `send_to_PC(OP_CALLEE, func_id, current_SP)` where `func_id` uniquely identifies a function. Then, the sanitizer plugin can track the call stack for each

thread. Combining the calling context and the static pointer ID, *IPEA-San* can uniquely identify temporary pointers.

Stack Objects. Stack objects are created by the `alloca` LLVM IR instruction. Similar to heap objects, we can instrument all the `alloca` IR instructions with an `OP_NEW` event to collect the stack object information. Similar to temporary pointers, stack objects are uniquely identified by a compile-time static ID.

Global Objects. In contrast to heap/stack objects, global objects are located in the `data` or `bss` sections with a lifespan of the entire firmware runtime. *IPEA-San* identifies every pointer reference to global objects and correspondingly inserts an instrumentation function `send_to_PC(OP_PROP, p_id, obj_id)`, where `p_id` is the ID of the receiving pointer and `obj_id` is a unique ID for the global object. For a global object, its location and size information is fixed. Therefore, we associate such information with its ID and pre-share it with the PC plugin beforehand.

Memory-Mapped Peripheral Objects. MCU vendors commonly integrate standard or custom-made peripherals, which are memory-mapped into the system memory at fixed locations. To access peripheral functions, developers typically define a data structure for each peripheral based on the register map. Then, a peripheral object can be instantiated by assigning a hardcoded address to a pointer of the corresponding data structure, as shown in the example below.

```
1  #define __IOM volatile
2  ...
3  typedef struct{
4    // Offset: 0x000 (R/W)  Interrupt Set Enable Register
5    __IOM uint32_t ISER[8U];
6          uint32_t RESERVED0[24U];
7      ...
8    // Offset: 0x300 (R/W)  Interrupt Priority Register
9    __IOM uint8_t  IP[240U];
10         uint32_t RESERVED5[644U];
11 }  NVIC_Type;
12 NVIC_Type *NVIC = (NVIC_Type *)0xE000E100UL;
```

Listing 2. Memory-mapped object for the NVIC peripheral.

IPEA-San identifies memory-mapped peripheral objects by detecting the assignments of constant non-memory addresses to pointers of data structure. Also, the fields in the data structure should have the `volatile` qualifier, a typical requirement when defining peripheral registers. These objects are treated as global objects and handled as mentioned before.

Intra-object Overflow Detection. If an object contains a sub-object inside, a buffer overrun of the sub-object can corrupt adjacent fields in the parent object, a problem known as intra-object overflow [75]. The pointer to a sub-object is typically derived by the *address-of* operator. For example, in the statement `q=&(p->a)`, `p` points to the parent object, `a` is the sub-object, and `q` is the derived pointer to the sub-object. To detect intra-object overflow, we need to associate a new tag for the sub-object and the derived pointer `q` so that `q` can be narrowed to stop the overflow. *IPEA-San* achieves this by inserting a function `send_to_PC(OP_SUB, q_id, p_id, p, offsetof(obj_type,a), sizeof(p->a))` at sub-object derivation sites. The five parameters represent the derived pointer, the pointer to parent object, the base of the parent object, the offset of the sub-object in the parent object, and the size of the sub-object, where the last three can be used to calculate the bounds of the sub-object.

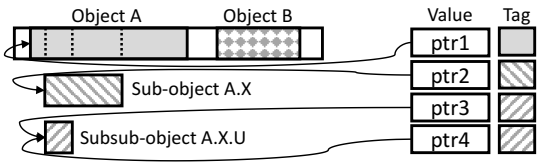

Fig. 3. Tag overlay to support intra-object overflow detection.

Metadata Maintenance. After an object creation event is received, the sanitizer plugin uses a monotonic counter to assign a tag to the range of the new object. This information is stored in a region of shadow memory where a byte on the MCU is mapped into four bytes on the PC. Therefore, our mechanism supports 32-bit tags (in contrast to 4-bit tags in MTE). We note that the memory size of an MCU is only hundreds of KB. Therefore, the PC has abundant resources to maintain the shadow memory.

The receiving pointer should be associated with the same tag. We implement it with a hash table that maps each pointer ID to its tag. Since the lifespan of a pointer differs, *IPEA-San* maintains two kinds of tables, namely *LocalID_Tag* table and *GlobalID_Tag* table for local and global pointers, respectively. The former table is allocated and maintained for each execution thread (e.g., a user task or interrupt handler) while the latter table is global which is shared among all execution threads. We collectively call both tables *ID_Tag* tables in this work.

On receiving an `OP_SUB` event, the sanitizer plugin first calculates the bounds of the sub-object. Then, the shadow memory is checked to make sure that this range has a tag that matches a bigger range (which should correspond to the parent pointer). If not, a memory bug is reported. Otherwise, the plugin assigns a new overlay tag to the corresponding shadow memory and the receiving pointer, a technique we termed as *tag overlay*. In essence, a byte can have multiple tags (e.g., for subsub-object) corresponding to different pointers that can access it. As shown in Figure 3, object `A` has a sub-object `A.X` which in turn has a subsub-object `A.X.U`. Depending on how a pointer is created, it can be assigned with different overlaid tags that distinguish the real bounds of the pointer (e.g., `ptr3` and `ptr4`). Note that a new tag is only assigned once for each sub-object. Future derived pointers of the same sub-object will reuse that tag.

2) Pointer Propagation: The value of a pointer can be passed to another one by assignment/arithmetic operations (intra-procedural flows) or function calls (inter-procedural flows). In both cases, the destination pointer should inherit the same tag as that of the source pointer.

Intra-procedural Flows. For each function, *IPEA-San* recognizes pointer propagation and inserts instrumentation as needed. As exemplified by lines 6/9 of Listing 1, the function `send_to_PC(OP_PROP, id_dst, id_src)` sends out the IDs for both the source pointer and the destination pointer.

Inter-procedural Flows. Pointers can be used as function arguments and return values. Therefore, inter-procedural prop-

agation should also be tracked. At each call site, *IPEA-San* parses the arguments and inserts a function with variable length argument: `send_to_PC(OP_PROP_CALLER, id1, ...)`, where all the pointer arguments are encoded in order. Correspondingly, in the prologue of the callee, `send_to_PC(OP_PROP_CALLEE, id1, ...)` is inserted, where all the receiving pointers are encoded in order.

If a function returns a pointer, *IPEA-San* sends out the ID of the returned pointer at epilogue using `send_to_PC(OP_PROP_RET, id)`. Correspondingly, after the call site, `send_to_PC(OP_PROP_RET_GET, id)` is inserted to inform the PC plugin of the receiving pointer.

Metadata Maintenance. The sanitizer plugin on PC simply retrieves the tag associated with the source pointer by looking up the *ID_Tag* tables and assigns it to the destination pointer. A new table entry is created for the destination pointer if needed.

3) Pointer Dereference: Pointer dereferences are considered to be the sinks of propagation. *IPEA-San* inserts a function `send_to_PC (OP_CHK, start, size, id)` before every pointer dereference. Pointer dereferences include not only the normal `load` and `store` IR instructions but also compiler intrinsics such as `memcpy` and `memmove`.

Metadata Maintenance. Upon receiving an `OP_CHK` event, the PC plugin retrieves the tag associated with the pointer by looking up *ID_Tag* tables. Then, it checks whether there exists a correct tag overlay for the memory range being accessed by looking up the shadow memory. If not found, an alert of possible memory error is triggered. Pointer dereferences do not change the metadata.

4) Object Deallocation: At function epilogues, *IPEA-San* inserts a function `send_to_PC(OP_RET, func_id)` to inform the PC of function returns. Before each deallocation operation such as `free`, *IPEA-San* inserts a function `send_to_PC(OP_FREE, id)` where `id` indicates the pointer for the freed object.

Metadata Maintenance. The `OP_RET` event informs the sanitizer plugin of the deallocation of local objects. The plugin then reclaims the corresponding entries in the *LocalID_Tag* table. The shadow memory for the affected local objects will also be marked as invalid by a special tag `0xFFFFFFFF`.

After receiving an `OP_FREE` event, the sanitizer plugin first verifies that there exists a valid entry indexed by `id` in *ID_Tag* tables. If not, an invalid free is detected. If the deallocation request is valid, the shadow memory of the object's range is marked invalid and the entry in the *ID_Tag* table is reclaimed. Note that there could exist other pointers which point to the now freed object. Dereferencing them will be detected by the tag matching mechanism described before (aka dangling pointer) and deallocating them again will be detected as an invalid free operation (aka double free).

5) Interrupt Support: IPEA-San is interrupt-aware. For the execution contexts of user tasks and interrupt handlers, *IPEA-San* maintains different *LocalID_Tag* tables. To track context switches, at each interrupt handler entry, an instrumentation function `send_to_PC(OP_IRQ, irq_num)` is inserted where `irq_num` indicates the target interrupt number. The interrupt return is reported by the `OP_RET` event of the corresponding interrupt handler.

Metadata Maintenance. A *LocalID_Tag* table is maintained for each execution context (i.e., a user task or interrupt handler). When an execution thread ends, the corresponding table is destroyed. For user tasks, the execution context is identified by the entry address of the task. For interrupts, the execution context is identified by the interrupt number.

D. Library Support

In addition to `memcpy` and `memmove`, string manipulation (e.g., `strcpy`, `strcat`, `sprintf`, etc.) is another category of memory access operations in `libc`. We provide wrappers for these functions so that necessary instrumentation is added before calling the actual implementations. Specifically, the wrapper determines the length of the string manipulation to be executed (by running `strlen`) and uses it to instrument memory checks for both the source and destination buffers.

For third-party libraries, developers need to write wrappers by themselves based on function semantics. If a function involves pointer dereferences, the corresponding pointer and the length of access should be identified. Then, an `OP_CHK` event can be inserted in the wrapper. If a function involves new object allocation, the resulting pointers should be identified, and then an `OP_NEW` event can be inserted. For example, the function `pxGetNetworkBufferWithDescriptor()` is used in the FreeRTOS TCP/IP stack to allocate new network buffers [6]. In the wrapper, we need to extract its size via the parameter `xRequestedSizeBytes` and the pointer value via the return value. Then an `OP_NEW` event can be inserted.

E. Optimizations

On top of the basic design, there are a few optimization opportunities to further improve *IPEA-San*.

Event Consolidation. For easy presentation, the instrumentation APIs summarized in Table I are organized based on the event semantics. However, many of these events can be consolidated into one to reduce the encoding overhead. For example, an `OP_PROP_CALLEE` event occurs when the callee function has pointer parameter(s). It can be combined with the `OP_CALLEE` event at the function prologue. The same applies to the `OP_IRQ` event, which occurs at the prologue of interrupt handlers. Likewise, events `OP_RET` and `OP_PROP_RET` can be consolidated at function epilogues.

Unnecessary Propagation. A substantial amount of instrumentation related to intra-procedural pointer propagation can be optimized out. Take the code in Listing 1 as an example. The tag of `p1` propagates to `p2`, which is eventually dereferenced at line 13. We can safely remove the instrumentation at line 6 and directly use `p1_id` to replace `p2_id` at line 12. This will not cause any side effects since the lifespan of `p2` is within the function. In contrast, we should never omit the instrumentation at line 9 since a global pointer may be used outside the current function. To remove unnecessary propagation, we first identify all the pointer dereferences. Then, an intra-procedural backward slicing is conducted to locate the source of the pointer, whose ID will be used in the `OP_CHK` instrumentation. Along the path of pointer propagation, no

event about pointer propagation will be instrumented unless the propagation target is a global pointer.

Unnecessary Initialization. Recall that in the basic design, the event about stack object creation is streamed out like heap objects. However, we found that this is unnecessary. In fact, the locations of stack objects are well determined given the base of the current stack frame. In particular, the DWARF-format debug information found in ELF binaries clearly specifies how to locate stack objects based on offsets. Since all the information produced during compilation can be shared with the PC, we can easily recover the locations of all stack objects based on the debug information and the base of the current stack frame. Note that the last parameter of the `OP_CALLEE` event indicates the current stack pointer at function entries.

VI. *IPEA-Fuzz*: A Fuzzer for MCU Firmware

IPEA-Fuzz is based on AFL [88] which uses edge coverage as feedback. We use stateless instrumentation to collect basic block transitions and run everything else on the PC.

A. IPEA-Fuzz Instrumentation

The original AFL injects the following code at the beginning of each basic block.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Here, `cur_location` is a random number generated at compile time to identify a basic block. The `shared_mem[]` array is a bitmap of 64 KB which holds in each byte the number of hits for a particular edge. In *IPEA-Fuzz*, we simply insert a function `send_to_PC(OP_BB, rand)` at the beginning of each basic block where `rand` is a compile-time random number.

Metadata Maintenance. After receiving an `OP_BB` event, the PC plugin follows the original AFL design to interface with `shared_mem[]` which is now stored on PC. Specifically, it uses the received random number to replace `cur_location` and emulates the code above as is. In this way, the original AFL is almost intact on the PC. The generated testcases are provided to the *IPEA* core so that they can be transmitted to the target MCU.

B. Interrupt Handling

An interrupt can kick in at any program point. This would lead to an excessive number of new edges that do not actually represent new firmware behaviors. To remove such noises, at each interrupt entry, the PC plugin creates a new context to calculate edge coverage. Specifically, on `OP_IRQ`, it saves the previous `prev_location` and uses a constant `0` as the new `prev_location`. When the interrupt returns, the saved `prev_location` is restored and the edge coverage calculation can be resumed from the old execution context.

TABLE II. Qualitative evaluation of *IPEA-San* in fault observation.

Memory Errors	*IPEA-San*	ASan
Stack-based Buffer Overflow	✓	✓
Heap-based Buffer Overflow	✓	✓
Global-based Buffer Overflow	✓	✓
Use-after-free	✓	✓
Double Free	✓	✓
Null Pointer Dereference	✓	✓
Peripheral-based Buffer Overflow	✓	✗
Intra-object Overflow	✓	✗

VII. Implementation

We have implemented a prototype of *IPEA*, *IPEA-San* and *IPEA-Fuzz* for Arm Cortex-M series MCUs. It supports any development board that is compatible with the SEGGER J-Link debug probe, the market-leading MCU debug solution [66]. Some development boards even have free on-board J-Link integration [67]. Seven different development boards—NXP FRDM-K64F (**K64F**, 1MB flash/256KB SRAM), FRDM-K66F (**K66F**, 2MB flash/256KB SRAM), LPCXpresso55S69 (**LPC55S69**, 640KB flash/320KB SRAM), STM32-NueloF411R (**STM32F4**, 512KB flash/128KB SRAM), STM32H7B3I-DK (**STM32H7**, 1MB flash/1MB SRAM), Raspberry Pi Pico (**RP2040**, 2MB flash/264KB SRAM) and nRF52-DK (**nRF52**, 192KB flash/24KB SRAM)—have been tested and evaluated.

Our firmware instrumentation module is based on LLVM 13.0.0. The collected data is encoded with a simple tag-value scheme where the tag takes one byte. All the optimizations mentioned in §V-E has been incorporated. We also developed a Python script based on `pyelftools` [11] to extract the static information from the compiled firmware. The *IPEA* core is developed using the SDK provided by SEGGER [69]. The *IPEA-San* plugin is written in C++ and the *IPEA-Fuzz* plugin is mainly inherited from AFL. The PC we used runs a Ubuntu OS and is equipped with an Intel Core i7-8750H CPU and 16 GB memory. In total, we contributed 4,555 lines of C/C++ code and 1,659 lines of Python code.

VIII. Evaluation

The main goal of our evaluation is to measure the benefit of offloading the analysis to PC. Since there is no fuzzer that can run entirely on the MCU, our case study is focused on *IPEA-San*. In addition, we are interested in the bug-finding capability of *IPEA-Fuzz*. Specifically, our evaluation aims to answer the following research questions. **RQ1:** What kind of memory errors can be captured by *IPEA-San*? **RQ2:** Can *IPEA-San* reduce resource consumption on MCUs? **RQ3:** Can the combination of *IPEA-San* and *IPEA-Fuzz* find bugs?

A. Sanitizer Capability

Before evaluating the benefit of the *IPEA* framework (§VIII-B), we first set up the comparison targets and measure the capability of *IPEA-San* in detecting memory corruption, including the type of errors it covers and the accuracy.

1) Comparison Targets: Finding an on-device sanitizer that runs out-of-box on MCUs is challenging due to the hardware/runtime differences. ASan, the state-of-the-art sanitizer officially supports user-space programs for PCs/Android and the Linux kernel (namely KASan [83]). Although there is an open-source port of ASan for MCUs [29], it is incomplete, merely implementing a wrapper for the heap manager to detect invalid heap accesses and a very basic memory access check mechanism. We improved it with support for stack and global objects and a more robust memory access check mechanism. Concretely, 11 callback functions were developed that hook on critical events including memory accesses and global objects initialization. Whenever possible, we kept the default configurations mentioned in the original paper [70]. However, some compromises and optimizations have to be made to accommodate MCU hardware. First, the original ASan dedicates 1/8 of memory to its shadow memory. This would map to 512 MB if the entire address space is covered, which is unacceptable. We made a compromise by limiting the protected address space to SRAM so that shadow memory is only needed for SRAM, similar to FuZZan [42]. For flash memory and MMIO regions, *IPEA-San* explicitly permits all accesses. For anything else, access is denied. This design saves memory but sacrifices fine-grained sanitization. For example, overflow to MMIO peripherals cannot be detected. However, this compromise is almost unavoidable since the peripheral region is typically too large to use shadow memory. Second, MCUs do not support virtual memory to map shadow memory to a non-existing page. We instead put shadow memory at the end of the SRAM region and explicitly deny accesses to it. Third, in the wrapper for the heap manager, the quarantine is implemented as a FIFO queue which can hold up to eight heap1 objects, in contrast to a fixed buffer in the original ASan.

Besides the *memory-optimized* ASan we ported, *µSBS* [63] is a sanitizer specifically designed for MCU firmware. It mimics ASan but is implemented via binary rewriting. However, it only detects overflows to heap objects, leaving stack and global objects unprotected. The main challenge lies in the difficulty of inferring the size and location of stack and global objects from binary. On the contrary, heap objects can be easily tracked by hooks to the allocator. Moreover, due to the avoidable run-time mapping table checkup, higher overhead is observed. Therefore, ASan's result is the upper limit that *µSBS* can theoretically achieve and we excluded *µSBS* from our evaluation.

2) Qualitative Evaluation: To qualitatively evaluate the types of memory corruption *IPEA-San* can detect, we made a toy firmware containing eight kinds of memory errors[1]: stack-based buffer overflow, heap-based buffer overflow, global-based buffer overflow, use-after-free, double free, null pointer dereference, peripheral-based buffer overflow, and intra-object overflow. Each time, an individual vulnerability was triggered based on a byte of the testcase.

Results. We instrumented the target firmware with ASan and *IPEA-San*. Then, multiple payloads with appropriate lengths were fed to the two firmware images to trigger a bug each time. Table II shows the results. Both solutions can detect all the traditional memory safety issues. However, only *IPEA-San* can detect peripheral-based buffer overflow and intra-object overflow.

[1]https://github.com/MCUSec/IPEA/blob/main/fw_samples/frdmk64f-sdk_v2.11.0/boards/frdmk64f/demo_apps/hello_world/hello_world.c

TABLE III. SELECTED TESTCASES IN JULIET. # TESTS INDICATES THE NUMBER OF SELECTED TESTCASES IN EACH CWE CATEGORY.

CWE Index	Description	# Tests
CWE121	Stack-based Buffer Overflow	2,432
CWE122	Heap-based Buffer Overflow	1,594
CWE124	Buffer Underwrite	682
CWE126	Buffer Overread	524
CWE127	Buffer Underread	682
CWE415	Double Free	190
CWE416	User After Free	118
CWE476	NULL Pointer Dereference	234
CWE761	Invalid Heap Pointer Free	152

3) Quantitative Evaluation: After knowing the general property of *IPEA-San*, we are interested in its correctness. More specifically, what are the rates of false positives (FP) and false negatives (FN)? In a related work (PACMem [51]), two benchmark suites were used to quantitatively evaluate the proposed sanitizer. They are the Juliet Test Suite [59] and Magma [39]. Unsurprisingly, both are geared towards Linux/Windows environments. We have attempted to port both for MCU but failed in porting Magma. The reason is that Magma contains many complex programs (e.g., OpenSSL) that cannot be accommodated on any MCU chip. In contrast, Juliet consists of many small programs exhibiting over 100 classes of errors. Therefore, we chose Juliet as the benchmark in our quantitative evaluation.

Juliet for MCUs. The Juliet Test Suite [59] provides a collection of testcases in C/C++ under 118 different CWEs. Each testcase contains both `BAD` and `GOOD` code. The `BAD` code contains the intended bug while the `GOOD` code is bug-free. Therefore, a testcase is always compiled into two programs (`BAD`+`GOOD`). The bug can be triggered in the `BAD` program without any input or by an input that satisfies a specific trigger condition. Since our focus is on memory safety issues, only relevant testcases were selected. We also excluded testcases written in C++ since we have not yet implemented wrapper functions for C++ STL containers. We leave C++ support (mostly engineering effort) as one of our future work. Similar to PACMem, we removed `CWE476_NULL_Pointer_Dereference__null_check_after_deref_*` from the benchmark. These tests perform null pointer checks after the pointer dereference, but without triggering any memory error. The selected tests are listed in Table III.

To accommodate these testcases for MCU, we made customization in three aspects. To the best of our investigation, this is the first benchmark for testing sanitizers on MCUs. 1) I/O Replacement: Juliet supports multiple input channels to trigger errors. However, these channels themselves do not exhibit any vulnerability. Since an MCU runtime may not provide all of these channels, we simply replaced them with a unified interface that accepts inputs from RTT without impacting the trigger conditions of bugs. 2) Feedback: We inserted a `BKPT` instruction with status code `0x01` into the `_exit()` function to inform analysis plugins on PC of the correct execution of the test. We also inserted a `BKPT` in-

TABLE IV. QUANTITATIVE EVALUATION WITH JULIET.

CWE Index	*IPEA-San*		ASan		# Tests w/o. ALLOCA	ASan	
	FN (%)	FP (%)	FN (%)	FP (%)		FN (%)	FP (%)
CWE121	0	0	1,220 (50.16%)	0	1,240	120 (9.63%)	0
CWE122	0	0	42 (2.26%)	0	1,594	42 (2.26%)	0
CWE124	0	0	170 (24.92%)	0	512	0	0
CWE126	0	0	179 (34.16%)	0	422	78 (18.32%)	0
CWE127	0	0	170 (24.92%)	0	512	0	0
CWE415	0	0	0	0	190	0	0
CWE416	0	0	0	0	118	0	0
CWE476	0	0	0	0	234	0	0
CWE761	0	0	0	0	152	0	0
Total	**0**	**%0**	**1,781 (26.95%)**	**%0**	**4,974**	**240 (4.83%)**	**%0**

struction with status code `0x02` into the `HardFault` handler to halt the execution when an unexpected error occurs. It catches corruptions not detected by sanitizers on a best-efforts basis. 3) Uncertainty Elimination: Some testcases rely on some randomnesses (e.g., an input of random number or uninitialized bytes on the stack) to trigger bugs. We manually analyzed these `BAD` programs and crafted inputs or modified the source code to reliably trigger the bugs.

Results. Having reliable inputs to trigger memory-related bugs in the `BAD` programs, a FN occurs when the sanitizer fails to report a memory corruption during the execution of a `BAD` program whereas a FP occurs when the sanitizer wrongly reports a non-existing memory corruption during the execution of a `GOOD` program. We used this metric to evaluate both *IPEA-San* and ASan. As summarized in Table IV, *IPEA-San* did not incur any FPs or FNs, and ASan incurred some FNs but not FPs. Although our result is promising, we note that in theory *IPEA-San* may also suffer from both FPs and FNs due to the unsound static analysis. The Juliet Test Suite—with enough sophistication and also used in related work to evaluate sanitizer capability [51]—just cannot trigger the deficiency of *IPEA-San*. We discuss more details about *IPEA-San* limitations in §IX. Below, we explain the root causes for FNs in ASan.

The FN rate of ASan (26.95%) is surprisingly higher than the one reported in the original paper [70]. After investigation, it turned out that many testcases in CWE121/124/126/127 use the `alloca` library function [1] (not confusing with the LLVM IR `alloca` instruction), which is not handled in KASan because the Linux kernel never depends on it. As a result, corruptions in these tests were not detected. For a fair comparison, we filtered out these tests and ran ASan against the remaining ones in a separate experiment. As shown on the right of Table IV, the FN rate of ASan drops into a normal range (4.83%). For the 240 FNs, 72 cases (e.g., `CWE121*__char_type_overrun*`) are due to intra-object overflow and 168 cases (e.g., `CWE121/126_fgets/fscanf/rand*`) are due to the well-known flaw of ASan in handling OOB accesses. In particular, it cannot detect non-linear OOB accesses that land in non-redzone locations. In these cases, the firmware uses inputs from users or random sources as indexes to access arrays. When the sanitizer failed, we also tried to enable the `HardFault` as a fall-back mechanism to capture unobserved system errors. This further brings the FN rate of ASan down to 3.38%.

4) Need for Better Sanitization: IPEA-San incurs less FNs than ASan mainly because it can reliably capture non-linear OOB accesses and intra-object buffer overflow. This section reviews previously reported CVEs in deeply embedded systems and measures the weight of these cases, aiming to estimate how *IPEA-San* can outperform ASan in revealing real bugs. We retrieved CVE reports related to deeply embedded system by searching for keywords of popular RTOSs (e.g., FreeRTOS, Contiki-NG) and chip vendors (e.g., NXP), resulting in a total of 37 CVEs. Note that we do not aim to collect a comprehensive list but rather try to make it representative by including diverse OS and chip vendors. For each CVE, we manually examined and classified it into either 1) a non-linear memory corruption, 2) a linear memory corruption, or 3) others (UAF, double free, integer error, etc.). We do not further distinguish CVE types in the "others" category since both *IPEA-San* and ASan perform similarly. Additionally, if a CVE also involves intra-object buffer overflow, we took a note. As shown in Table X, out of 37 CVEs, the numbers of linear and non-linear memory corruption are 11 and 13 respectively, and there is 1 intra-object buffer overflow. This ratio of linear to non-linear memory corruption is in line with a technique report released by Microsoft [52], which states that non-linear memory corruption has surpassed linear memory corruption. This indicates that *IPEA-San* should perform better in 37.84% (14/37) of the collected CVEs. We emphasize that it does not mean ASan can never find these bugs. For non-linear memory corruption, it just depends on whether the target address lands in a redzone or a non-redzone. However, ASan can never find CVE-2021-42553, which is an intra-object buffer overflow bug.

B. Overhead of IPEA-San

We used 12 MCU applications and a popular benchmark for embedded platforms called BEEBS [61] to evaluate the memory and run-time performance overhead of *IPEA-San*. The 12 MCU applications were collected from multiple sources including demos from chip SDKs [60], [77], open-source projects [7], [9], [12] and related work [28], [73], presenting a variety of application scenarios with different complexities.

- `PinLock` simulates a smart lock by accepting user PIN via UART and verifying its SHA-256 hash value.
- `CNC` is based on the GRBL-Advanced project [7] that implements a CNC (computer numerical control) milling machine. It accepts g-code commands from a workstation that runs Candle 2 [65] to control the stepping motors.
- `nRF52-Keyboard` allows nRF52 series MCUs to power Bluetooth keyboards.
- `ClockAndWeather` invokes open APIs [15], [13] to retrieve the time and weather forecast for up six cities via WiFi.
- `AudioPlayer` plays an audio file stored in the SD card.
- `WeighScale` is a weigh scale application that collects and transfers users' healthy data to the PC via the USB port.
- `HttpServer` implements an HTTP server on top of lwIP TCP/IP stack [34].
- `U-Disk` turns an MCU into a USB disk which can be enumerated by PC.
- `MQTT-Echo` integrates the AWS MQTT library and implements an MQTT client to publish and echo messages on a specific topic.
- `App-Scheduling`, `App-Timers` and `App-IRQs` are three template applications for quickly prototyping

TABLE V. NORMALIZED FLASH/RAM OVERHEAD ON SELECTED BEEBS PROGRAMS AND REAL-WORLD FIRMWARE (THE HIGHER, THE WORSE).

Firmware	Baseline					*IPEA-San*						ASan					
	Flash (Bytes)	Stack (Bytes)	Heap (Bytes)	Global (Bytes)	Total RAM (Bytes)	Flash (x)	Stack (x)	Heap (x)	Global (x)	RTT† (Bytes)	Total RAM (x)	Flash (x)	Stack (x)	Heap (x)	Global (x)	Shadow‡ (Bytes)	Total RAM (x)
aha-compress	4,356	112	-	184	296	1.81	1.14	-	1.00	1,076	4.69	1.94	3.00	-	1.63	2,085	9.19
ctl-stack	4,796	120	816	204	1,140	2.12	1.13	1.00	1.00	1,076	1.96	1.91	2.80	3.13	3.92	2,148	5.12
ctl-string	8,076	104	232	212	548	1.90	1.00	1.00	1.00	1,076	2.97	1.62	2.00	3.03	5.83	2,202	7.94
frac	7,456	240	-	268	508	1.46	1.03	-	1.00	1,076	3.14	1.52	1.60	-	1.64	2,103	5.76
huffbench	4,960	7,824	1,001	188	9,013	2.08	1.01	1.00	1.00	1,076	1.13	2.00	1.13	4.60	2.17	2,099	1.77
sglib-hashtable	6,408	224	800	664	1,688	2.12	1.00	1.00	1.00	1,076	1.64	1.63	2.07	2.14	1.39	2,163	3.12
Geo. Mean	-	-	-	-	-	**1.90**	-	-	-	-	**2.32**	**1.76**	-	-	-	-	**4.78**
PinLock	20,956	416	-	760	1,176	1.67	1.06	-	1.00	1,076	1.94	3.88	1.81	-	72.33	8,919	54.97
CNC	74,800	344	-	12,708	13,052	1.44	1.17	-	1.00	1,076	1.09	1.37	1.51	-	2.00	3,684	2.27
nRF52-Keyboard	52,428	478	-	6,224	6,702	2.34	1.05	-	1.09	1,076	1.25	1.70	2.17	-	1.80	1,540	2.05
ClockAndWeather	286,184	732	5,456	179,164	185,360	1.21	2.03	1.00	1.00	1,076	1.01	1.30	1.57	1.01	1.84	45,390	2.06
AudioPlayer	101,832	872	-	13,992	14,864	1.56	1.11	-	1.07	1,076	1.14	1.39	2.26	-	4.01	7,272	4.40
WeighScale	24,052	768	-	21,060	21,828	2.79	1.66	-	1.02	1,076	1.01	2.07	2.32	-	2.86	7,754	3.19
HttpServer*	71,932	840	-	65,408	66,248	2.67	1.05	-	1.01	1,076	1.03	-	-	-	-	-	-
U-Disk*	33,356	188	12,168	44,632	56,988	2.75	1.96	1.00	1.00	1,076	1.02	-	-	-	-	-	-
MQTT-Echo*	63,976	632	10,436	59,020	70,088	2.56	1.40	1.00	1.00	1,076	1.02	-	-	-	-	-	-
App-Scheduling	355,108	124	-	7,764	7,888	1.10	1.22	-	1.00	1,076	1.13	1.09	1.42	-	1.32	1,250	1.48
App-Timers	346,816	112	-	7,696	7,808	1.11	1.21	-	1.00	1,076	1.14	1.08	1.42	-	1.32	1,250	1.48
App-IRQs	356,588	108	-	7,800	7,908	1.12	1.17	-	1.00	1,076	1.14	1.07	1.41	-	1.29	1,250	1.45
Geo. Mean	-	-	-	-	-	**1.74**	-	-	-	-	**1.14**	**1.52**	-	-	-	-	**3.06**

†: RTT buffer plus other metadata. ‡: Shadow memory plus other metadata. *: Failed to compile using ASan.

projects based on FreeRTOS for the popular Raspberry Pi RP2040 MCUs.

TABLE VI. TESTED REAL-WORLD APPLICATIONS

Application	Target MCU	OS	Source
PinLock	K64F	Bare-metal	[28]
CNC	STM32F4	Bare-metal	[7]
nRF52-Keyboard	nRF52	Bare-metal	[9]
ClockAndWeather	STM32H7	FreeRTOS	[76]
AudioPlayer	K66F	Bare-metal	NXP SDK [60]
WeighScale	K66F	Bare-metal	NXP SDK [60]
HttpServer	K66F	Bare-metal	NXP SDK [60]
U-Disk	K64F	FreeRTOS	NXP SDK [60]
MQTT-Echo	K64F	FreeRTOS	NXP SDK [60]
App-Scheduling	RP2040	FreeRTOS	[12]
App-Timers	RP2040	FreeRTOS	[12]
App-IRQs	RP2040	FreeRTOS	[12]

The names of these applications along with their target MCU, OS type, and source are shown in Table VI. Our comparison targets include ASan and a baseline build without any instrumentation. All the samples were compiled under the `-Os` optimization level which is the de facto option adopted in embedded systems. For *IPEA-San*, we used a RTT buffer of 1 KB. This was chosen empirically to strike a balance between performance and memory consumption. For ASan, we used the same default configuration as mentioned in §VIII-A. In Table V, we list the results of 12 applications and 6 randomly selected BEEBS programs. Besides collecting performance data, an immediate observation after running BEEBS with *IPEA-San* is that we found two "silent" memory corruptions in `qsort` and `select`. Later investigation shows that these programs wrongly initialize an index, leading to OOB array accesses.

1) Flash Overhead: Flash is commonly used by MCU devices to store code and initialized global data. The code stays in the flash throughout the execution, while the initialized global data is copied to the SRAM during bootstrapping. As shown in Table V, the flash consumption of *IPEA-San* is on a par with ASan. On the one hand, *IPEA-San* consumes less flash for initialized global data since there is no need to insert redzones around global objects. On the other hand, it adds more code to track the run-time information (e.g., pointer creation/propagation) while ASan only adds code to check pointer dereferences. We also found that the code increment is mainly introduced by pointer dereferences, which is in line with existing studies [90]. Although 1.74x overhead seems high, MCU chips typically have abundant flash memory to tolerate it. First, flash is less expensive than SRAM [14] and therefore its capacity is much larger on typically MCU chips. We dumped the configuration information of the first 500 MCU chips returned from the DigiKey website[2]. The average flash and SRAM sizes are 374KB and 94KB respectively. Second, firmware does not consume a high amount of flash. For example, we compiled all the demo programs shipped with the K64F SDK [60] and calculated the maximum flash/SRAM consumption in each application category. As shown in Table VII, the maximum flash usage is only 36.94% whereas many samples use more than 50.00% of total SRAM.

TABLE VII. MAXIMUM FLASH/SRAM CONSUMPTION OF K64F SDK DEMOS IN EACH APPLICATION CATEGORY.

Category	Flash Usage (%)	SRAM Usage (%)
fmstr	5.21	11.22
aws	21.96	76.33
azure	28.44	98.66
bootloader	3.49	3.52
emWin	12.69	22.51
littlevgl	18.90	32.26
lwip-https	22.15	80.45
mmcau	2.54	5.32
sdcard	3.23	21.20
se_hostlib	**36.94**	**99.61**
secure-subsystem	1.02	21.14
usb-device	2.68	22.43
usb-host	2.99	22.70

2) SRAM Overhead: SRAM is used by MCUs to store global data (initialized and uninitialized), stack and heap. While the size of global data can be determined statically (`.data` segment plus `.bss` segment), the stack/heap usage has to be measured dynamically at run-time.

The total SRAM consumption includes the stack/heap/-global usage as well as the respective metadata. For *IPEA-San*

[2]https://www.digikey.com/en/products/filter/embedded/microcontrollers/685

and ASan, the metadata mainly comes from the 1-KB RTT buffer and the 8-to-1 shadow memory, respectively. As shown in Table V, benefiting from offloading analysis tasks to the PC, *IPEA-San* significantly reduces the SRAM overhead compared with ASan. For all the samples, there is no SRAM overhead on heap and global data. For stack, the maximum overhead is 2.79x in `WeighScale`. The increased stack consumption is mainly used for storing and passing arguments of the instrumented calls. In contrast, the SRAM overhead of ASan is much higher due to the inserted redzones and proportional overhead on shadow memory. By choosing smaller redzones, the SRAM overhead of ASan might be reduced. However, this will inevitably increase the FN rate. Our choice of redzone size follows the default setting in the original paper [70]. We also observed an anomaly in `PinLock` where the overhead on global data reaches 72.33x. It turned out that this program uses the `mbedtls` library to calculate SHA-256, which defines an excessive number of global objects.

Regarding the total SRAM overhead, *IPEA-San* incurs about 2.32x overhead compared with ASan at 4.78x for the BEEBS benchmark. This margin further increases for real-world applications (1.14x vs. 3.06x). This is because the baseline SRAM consumption in BEEBS programs is insignificant compared with the constant RTT overhead in *IPEA-San*. However, as the baseline increases in real-world applications, the real SRAM overhead becomes dominating. As a result of ASan's demanding SRAM requirement, we failed to compile `U-Disk`, `MQTT-Echo` and `HttpServer` with ASan after allocating the minimally needed SRAM. Note that these three programs are part of the official SDK shipped with the COTS MCU boards.

3) Performance Overhead: In Appendix §A-B, we show the run-time performance overhead. *IPEA-San* incurs slightly higher overhead (86% vs. 67% slowdown in the BEEBS benchmark and 14% vs. 4% slowdown in real-world applications). Again, this is because *IPEA-San* adds more code to track the run-time information than ASan does. Also, transmitting the collected events via RTT takes time. For real-world programs that have more I/O operations, the overhead of *IPEA-San* is only 14%. Interestingly, we observed that ASan for MCUs incurs less performance overhead compared with ASan for traditional software. Specifically, the original ASan paper reported 73% slowdown [70] while an independent study reported 107% slowdown [90]. Other than the optimization we made to ASan for MCUs (see VIII-A1), we attribute this partially to the fact that MCU firmware is not subject to the heavy memory management overhead in traditional ASan [42].

C. Fuzzing Evaluation

This section first presents the results of using *IPEA-Fuzz* plus *IPEA-San* to find memory-related bugs in MCU firmware. Then, we provide experimental evidence of the indispensability of real hardware for testing complex driver code.

1) Bug Finding Capability: We selected two groups of fuzzing targets. The IoT library group includes a JPEG decoder [8], a PNG decoder [10], an XML parser [55] and the toy example we introduced earlier. The first two are popular projects specifically optimized for Arduino devices. The XML parser is based on the Expat project [5]. We reused the sample provided in a related work [55] which injected six artificial memory bugs into Expat. The peripheral driver group targets driver code for four peripherals—UART, USB, WiFi and microSD. The driver code for UART and WiFi are part of `PinLock` and `ClockAndWeather`, respectively. For USB, we tested three distinct instances using different boards. They are the USB stack shipped with NXP SDK [60] running on K64F/K66F, the USB stack shipped with STM32 SDK [77] running on STM32H7, and the proprietary emUSB stack [4] provided by SEGGER running on LPC55S69. It is worth noting that samples in the IoT library group are largely hardware independent and therefore can be tested more efficiently by emulating the compiled binary (e.g., using Fuzzware [64]). When developers can devote time to porting them from the source code, they can even be tested natively on a PC. On the other hand, it is extremely difficult if not impossible to rehost samples in the peripheral driver group, making real hardware essential for testing peripheral driver (see §VIII-C2).

TABLE VIII. BREAK-DOWN OF INDIVIDUAL TESTCASE EXECUTIONS

Firmware	Reset (ms)	FuzzStart (ms)	Exec (ms)	Analysis (ms)	Total (ms)
Toy	-	1.00	2.00	1.00	4.00
Expat XML	-	1.00	56.00	1.00	58.00
JPEGDEC	-	1.00	102.00	1.00	104.00
PNGdec	-	1.00	97.00	1.00	99.00
UART	70.00	1.00	121.00	1.00	193.00
WiFi	70.00	1.00	3,831.00	3.12	3,905.00
USB Host (K64F/K66F)	70.00	1.00	2,682.00	1.00	2,755.00
USB Host (STM32H7)	71.00	1.00	2,182.00	2.00	2,356.00
emUSB-Host (LPC55S69)	70.00	1.00	1,912.00	1.00	1,984.00
microSD	70.00	1.00	2,682.00	1.00	2,754.00

Two fuzzing modes were used in our evaluation. In normal mode, we reset the board for each testcase execution. In *persistent mode*, a single long-lived execution is reused to try out multiple testcases without resetting the hardware every time. The normal mode is necessary for fuzzing peripheral drivers because we must ensure a clean hardware state for each run. The persistent mode on the other hand can be beneficial for fuzzing samples in the IoT library group since the hardware state is unlikely to be faulty after running a testcase. By testing the toy example, we found that persistent mode can substantially improve fuzzing speed. Specifically, using the same hardware, persistence-mode fuzzing reached 276.43 executions/second while the normal mode only reached 12.95 executions/second.

To explain this, we measured a time break-down of each testcase execution, as shown in Table VIII. For each sample, we selected a "normal" testcase that drived execution along a typical execution path and measured the time spent on each stage. *Reset* refers to the time for resting the hardware. *FuzzStart* indicates the time for the *IPEA* core to prepare and transmit a testcase to the target device. *Exec* is the execution time on the target device. *Analysis* is the time for the *IPEA* plugins to analyze the received data on PC. As shown in the table, most of the time is spent on hardware reset and testcase execution. While the reset time is almost constant (70 ms), the execution time heavily depends on the target firmware. For IoT libraries that run fast, the benefit of removing the reset overhead can be significant, as shown in the toy example. Here, the toy firmware only took 2 ms for execution. Therefore, 276.43 executions/second can be viewed as the upper limit that our prototype can reach. In contrast, it took much longer to

TABLE IX. FUZZING RESULTS

Firmware	Target MCU	Time (s)	# Execution	Exec/sec	Paths	Crashes/Hangs
Toy	K64F	7,200	1,990,296	276.43	9	8/0
Expat XML	STM32H7	86,400	1,539,648	17.82	1,098	5/0
JPEGDEC	STM32H7	83,752	805,308	9.61	903	17/4
PNGdec	STM32H7	84,011	848,512	10.10	1,001	22/10
UART	K64F	86,471	537,849	6.22	22	0/0
WiFi	STM32H7	151,212	32,218	0.21	153	1/5
USB Host	K64F	327,910	127,884	0.39	99	22/47
USB Host	STM32H7	347,774	141,277	0.45	96	0/57
emUSB-Host	LPC55S69	122,452	87,619	0.88	106	3/23
microSD	K64F	86,457	30,240	0.35	77	0/6

complete a test for peripheral driver. This is because driver code needs to deal with hardware activities which often involve delays. For example, we observed that the USB driver spent 1.5∼2 seconds to perform enumeration, while the WiFi driver spent 4∼5 seconds to connect to the Access Point. In these cases, the reset overhead can be largely amortized.

In Table IX, we show the results of fuzzing IoT libraries in persistent mode and driver code in normal mode. Most samples were fuzzed for 24 hours. However, to make meaningful results, some samples received more time depending on the fuzzing speed. *IPEA-Fuzz* found all eight bugs in the toy firmware and five out of six bugs in Expat XML parser. The missing one is a format string vulnerability [55] which is not checked in *IPEA-San*. For JPEGDEC, we found three new global buffer overflow bugs. For PNGdec, *IPEA-Fuzz* reported one known CVE [3]. In addition, *IPEA-Fuzz* not only reproduced two CVEs reported in μAFL [50] but also found one new UAF bug in the NXP USB driver (**NXP-USB**). It is worth noting that this sample has been extensively tested in μAFL. Moreover, we found a new buffer overflow bug in the WiFi driver shipped with the STM32 SDK (**STM32-WiFi**), a new Denial-of-Service (DoS) bug in the STM32 USB stack (**STM32-USB**), and a buffer overflow bug in SEGGER emUSB-Host (**SEGGER-USB**). All the bugs have been reported to and confirmed by the respective maintainers. Patches for JPEGDEC have been merged to the main repository. **NXP-USB** and **SEGGER-USB** have been fixed by the maintainers and the patched code will be released with the next distributions. We are working with ST to fix the two remaining bugs. The bug details for them are provided in Appendix §A-C.

2) Indispensability of Real Hardware for Testing Driver Code: Low-level driver code frequently interacts with peripherals whose behavior is hard to model, leading to challenges in emulation-based firmware testing. Although existing work has made substantial progress [32], [23], [91], [64], [44], none of them can handle complex peripherals. We demonstrate this by testing four samples in the peripheral driver group (WiFi, NXP USB, STM32 USB, emUSB) using a state-of-the-art emulation-based solution named Fuzzware [64]. During fuzzing, we monitored the execution trace to check if Fuzzware can properly initialize USB/WiFi. Unfortunately, this never happened during the two-day testing. Sometimes Fuzzware gave up emulation because it can never consume the testcase within a predefined number of basic block execution. The STM32 USB sample was even stuck during booting. It turned out that Fuzzware failed to generate authentic MMIO responses for the RCC peripheral in the function `SystemClock_Config()`. This explains why existing solutions sometimes need manual removal of hard-to-emulate logic in the source code[3]. In contrast, *IPEA* enables streamlined firmware testing with high fidelity, thanks to the involvement of real hardware.

IX. DISCUSSION

***IPEA-San* Limitations.** Although our evaluation shows promising results, *IPEA-San* has limitations in handling several corner cases due to the unsoundness of static analysis, resulting in both FNs and FPs. First, although we use typecast instructions (`ptrtoint`, `inttoptr`, `bitcast`) to extensively find and infer all pointers, some pointers—defined as integers inside a data structure—cannot be easily recovered. Our static analysis may miss implicit pointer propagation that comes with such in-structure integer-cast pointers, leading to FNs. Second, in some network protocol parsers, a zero-length array (e.g., `payload[0]`) may be placed at the end of a structure that is really a header for a variable-length object [2]. The empty array only serves as a placeholder for future payload to receive. Such "struct hack" can lead to FPs since our intra-object overflow detection can only tag the original data structure where the payload is considered empty.

Extension for Other Analysis Techniques. As a framework, *IPEA* is not limited to the analysis we prototyped in this work. By streaming out more information, other analysis plugins can be supported. For example, by including type information in the per-pointer metadata, our current *IPEA-San* prototype can be extended to detect type confusion errors in C++ code, similar to existing type sanitizers [38], [26], [41]. Specifically, the `OP_NEW` event will carry a type ID of the object. On receiving it, the PC plugin can associate a run-time type data structure to the pointer, which encodes all permissible casts for that pointer. When typecasting occurs, a new event will be streamed out and run-time type checking can be performed. In addition, many other sanitizers (e.g., MemorySanitizer to detect uninitialized reads [80]) can be supported by streaming out the respective run-time information.

X. RELATED WORK

Dynamic Firmware Analysis. Analyzing MCU firmware faces many unique challenges, drawing research efforts from different angles. On-device methods conduct analysis directly on the target hardware. For example, μAFL [50] fuzzes peripheral driver code with the help of a debug dongle and utilizes the Arm ETM debug feature to collect the instruction trace as fuzzer feedback. A similar idea adopting the SWO debug feature is proposed by Beckmann et al. [21]. However, these solutions are platform-specific and only support fuzzing. Over-the-air fuzzing has been explored to find bugs in Bluetooth controllers [36], [35]. However, they only collect coarse-grained execution information by observing I/O and require domain knowledge to define faulty conditions. In contrast, *IPEA* targets general firmware testing problems.

Although on-device analysis provides high fidelity, it is not scalable. To address this issue, rehosting-based approaches either emulate the firmware binary on a PC [32], [23], [91], [64] or port the application from source code [49]. Rehosting

[3]https://github.com/fuzzware-fuzzer/fuzzware-experiments/blob/main/03-fuzzing-new-targets/contiki-ng/building/patches/cc2538_read.patch

can be more efficient in firmware testing, provided that the underlying hardware is accurately modeled. Unfortunately, this is not the case since it is very challenging to model the behavior of complex peripherals, as indicated in literature [31], [85] and our experiments (§VIII-C2). With the compromise of not covering low-level code, HALucinator [24] side-steps the driver emulation problem by simulating hardware abstraction layers (HALs) on the host. This approach only covers the upper application logic and does not work for firmware that does not use HALs. In contrast, *IPEA-San* is a full-stack solution.

In the middle ground, hardware-in-the-loop rehosting redirects I/O interactions to the physical hardware [87], [54], [47], [25]. Frankenstein [62] directly uses dumped firmware images from real devices to re-establish emulator states. These solutions leverage real hardware to improve emulation fidelity. *IPEA* leverages real hardware to collect analysis-specific information directly from the target.

Remote Attestation. Collaboratively running an analysis on MCU and PC has been explored in C-FLAT [16] and OAT [78]. These systems enable remote attestation of control-flow paths for IoT devices. This is achieved by using TrustZone to sign the hash of the abstract execution path and send the result to a remote verifier. Technically, *IPEA* also streams internal execution information to a PC. However, *IPEA* is geared towards general firmware testing during development and thus incurs less restriction compared to C-FLAT and OAT. Concretely, a remote attestation system must consider the transmission overhead, limiting its current application to control-flow integrity (CFI) where only the signed hash of the execution path needs to be transmitted. To enable remote attestation of memory safety, finding an efficient strategy to encode the collected information is a challenge, which can be interesting for future research.

Sanitizers. Sanitizers are commonly used in software testing. Besides addressability bugs, they are also used to detect concurrency bugs [81], undefined behavior [82], uninitialized reads [80], type confusion [38], etc. Regarding addressability-oriented sanitizers, there are two popular designs. Redzone-based approaches [70], [63], [86] place redzones as invalid memory around objects. Then, shadow memory is used to track the status of each byte and an alert is raised when an invalid byte in redzone is accessed. Pointer-based approaches [56], [57], [58], [19], [27], [43] encode pointer capabilities (i.e., which object it can access) into the pointers. The capability is either encoded as a fat pointer representation [58] or in disjoint metadata [56], [57].

The memory overhead introduced by ASan transforms to significant performance slowdown during sanitizer setup/teardown [42]. Therefore, recent work tries to reduce metadata with optimized data structure [42], [20] or by leveraging hardware features [79], [51], [89]. ASan for MCUs ported in this work actually incorporates the idea of FuZZan [42] to reduce the address space of the target firmware. *IPEA-San* further eliminates metadata in firmware with decoupled design. Using static analysis, existing work also reduces performance overhead by removing unnecessary checks [90]. It is orthogonal to *IPEA-San* and can be integrated.

XI. Conclusions

We present the design and implementation of the *IPEA* firmware analysis framework and two analysis plugins for it. They seamlessly integrate into existing firmware development environments, allowing developers to run advanced firmware testing while developing firmware. By offloading analysis to the development PCs, the proposed analysis techniques significantly reduce memory overhead compared with solutions that run entirely on MCUs. Using our prototype to test real-world firmware samples, we found seven zero-day bugs.

Acknowledgment

The authors would like to thank the anonymous reviewers for their extensive feedback on earlier drafts of this paper. This work was supported by the US National Science Foundation under grant CNS-2238264, Cisco Research, and a grant from the University of Georgia Research Foundation, Inc. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] "alloca(3) — Linux manual page," https://man7.org/linux/man-pages/man3/alloca.3.html, (Retrieved: 04/12/2023).

[2] "Arrays of Length Zero," https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html, (Retrieved: 04/12/2023).

[3] "CVE-2022-35011," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-35011, (Retrieved: 04/12/2023).

[4] "emUSB-Host — USB peripherals with embedded devices," https://www.segger.com/products/connectivity/emusb-host/, (Retrieved: 04/12/2023).

[5] "Expat XML parser," https://libexpat.github.io, (Retrieved: 04/12/2023).

[6] "FreeRTOS-Plus-TCP documentation: pxGetNetworkBufferWithDescriptor()," https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/API/pxGetNetworkBufferWithDescriptor.html, (Retrieved: 04/12/2023).

[7] "Grbl-Advanced CNC milling," https://github.com/Schildkroet/GRBL-Advanced, (Retrieved: 04/12/2023).

[8] "JPEGDEC," https://github.com/bitbank2/JPEGDEC, (Retrieved: 04/12/2023).

[9] "nrf52-keyboard," https://github.com/Lotlab/nrf52-keyboard, (Retrieved: 04/12/2023).

[10] "PNGdec," https://github.com/bitbank2/PNGdec, (Retrieved: 04/12/2023).

[11] "pyelftools," https://github.com/eliben/pyelftools, (Retrieved: 04/12/2023).

[12] "RP2040-FreeRTOS," https://github.com/smittytone/RP2040-FreeRTOS, (Retrieved: 04/12/2023).

[13] "Weather API," https://openweathermap.org/api, (Retrieved: 04/12/2023).

[14] "what is the difference between Flash memory and SRAM," https://forum.43oh.com/topic/7661-what-is-the-difference-between-flash-memory-and-sram/, (Retrieved: 04/12/2023).

[15] "WorldTimeAPI," https://worldtimeapi.org/, (Retrieved: 04/12/2023).

[16] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[17] Arm Holdings, "Armv8.5-A Memory Tagging Extension," https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf, (Retrieved: 04/12/2023).

[18] ——, "What is semihosting?" https://developer.arm.com/documentation/dui0375/g/What-is-Semihosting-/What-is-semihosting-, (Retrieved: 04/12/2023).

[19] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, 1994, pp. 290–301.

[20] J. Ba, G. J. Duck, and A. Roychoudhury, "Efficient greybox fuzzing to detect memory errors," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[21] M. Beckmann and J. Steffan, "Coverage-guided fuzzing of embedded systems leveraging hardware tracing," in *Computer Security. ESORICS 2022 International Workshops*. Cham: Springer International Publishing, 2023, pp. 362–378.

[22] G. Beniamini, "Over The Air: Exploiting Broadcom's Wi-Fi Stack," https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, (Retrieved: 04/12/2023).

[23] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20, 2020.

[24] Clements, Abraham and Gustafson, Eric and Scharnowski, Tobias and Grosen, Paul and Fritz, David and Kruegel, Christopher and Vigna, Giovanni and Bagchi, Saurabh and Payer, Mathias, "Halucinator: Firmware re-hosting through abstraction layer emulation," 2020.

[25] N. Corteggiani and A. Francillon, "Hardsnap: Leveraging hardware snapshotting for embedded systems security testing," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 294–305.

[26] G. J. Duck and R. H. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 181–195.

[27] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *NDSS*, 2017.

[28] Embedded Security, "PinLock," https://github.com/embedded-sec/ACES/tree/master/test_apps/pinlock, (Retrieved: 04/12/2023).

[29] Erich Styger, "Finding Memory Bugs with Google Address Sanitizer (ASAN) on Microcontrollers," https://mcuoneclipse.com/2021/05/31/finding-memory-bugs-with-google-address-sanitizer-asan-on-microcontrollers/, (Retrieved: 04/12/2023).

[30] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier," *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.

[31] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, "Sok: Enabling security analyses of embedded systems via rehosting," 2021.

[32] B. Feng, A. Mera, and L. Lu, "P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[33] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *2020 IEEE Secure Development Conference (SecDev)*, 2020, pp. 23–30.

[34] Free Software Foundation, Inc., "lwIP A Lightweight TCP/IP stack," https://savannah.nongnu.org/projects/lwip/, (Retrieved: 04/12/2023).

[35] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "Braktooth: Causing havoc on bluetooth link manager via directed fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022.

[36] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "Sweyntooth: Unleashing mayhem over bluetooth low energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 911–925.

[37] F. Gritti, F. Pagani, I. Grishchenko, L. Dresel, N. Redini, C. Kruegel, and G. Vigna, "Heapster: Analyzing the security of dynamic allocators for monolithic firmware images," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1559–1559.

[38] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 517–528. [Online]. Available: https://doi.org/10.1145/2976749.2978405

[39] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[40] J. Homan, S. McBride, and R. Caldwell, "IRONGATE ICS Malware: Nothing to See Here...Masking Malicious Activity on SCADA Systems," https://www.fireeye.com/blog/threat-research/2016/06/irongate_ics_malware.html, (Retrieved: 04/12/2023).

[41] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "Hextype: Efficient detection of type confusion errors for c++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2373–2387.

[42] Y. Jeon, W. Han, N. Burow, and M. Payer, "Fuzzan: Efficient sanitizer metadata design for fuzzing." in *USENIX Annual Technical Conference*, 2020, pp. 249–263.

[43] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[44] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson

[45] O. Karliner, "FreeRTOS TCP/IP Stack Vulnerabilities – The Details," https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/, (Retrieved: 04/12/2023).

[46] M. Kol and S. Oberman, "19 Zero-Day Vulnerabilities Amplified by the Supply Chain," *JSOF, White Paper*, (Retrieved: 04/12/2023).

[47] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher

[48] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid," *Electricity Information Sharing and Analysis Center (E-ISAC)*, vol. 388, 2016.

[49] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From library portability to para-rehosting: Natively executing open-source microcontroller oss on commodity hardware," in *28th Network and Distributed System Security Symposium*, ser. NDSS '21. The Internet Society, 2021.

[50] W. Li, J. Shi, , F. Li, J. Lin, W. Wang, and L. Guan, "μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware," in *2022 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*. IEEE, 2022.

[51] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1901–1915.

[52] M. Matt, "Trends and challenges in the vulnerability mitigation landscape," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, 2019.

[53] MSRC Team, "BadAlloc – Memory allocation vulnerabilities could affect wide range of IoT and OT devices in industrial, medical, and enterprise networks," https://msrc-blog.microsoft.com/2021/04/29/badalloc-memory-allocation-vulnerabilities-could-affect-wide-range-of-iot-and-ot-devices-in-industrial-medical-and-enterprise-networks/, (Retrieved: 04/12/2023).

[54] M. Muench, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *Workshop on Binary Analysis Research*, 2018.

[55] 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 *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[56] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in

Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009, pp. 245–258.

[57] ——, "Cets: compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, 2010, pp. 31–40.

[58] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, p. 477–526, May 2005. [Online]. Available: https://doi.org/10.1145/1065887.1065892

[59] NSA Center for Assured Software, "Juliet C/C++ 1.3," https://samate.nist.gov/SARD/test-suites/112, (Retrieved: 04/12/2023).

[60] NXP, "MCUXpresso SDK Builder," https://mcuxpresso.nxp.com/en/welcome, (Retrieved: 04/12/2023).

[61] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[62] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36.

[63] M. Salehi, D. Hughes, and B. Crispo, "μSBS: Static binary sanitization of bare-metal embedded devices for fault observability," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 381–395.

[64] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[65] Schildkroet, "Candle 2," https://github.com/Schildkroet/Candle2, (Retrieved: 04/12/2023).

[66] SEGGER, "J-Link Debug Probes," https://www.segger.com/products/debug-probes/j-link/, (Retrieved: 04/12/2023).

[67] ——, "J-Link OB - The On-Board Debug Probe," https://www.segger.com/products/debug-probes/j-link/models/j-link-ob, (Retrieved: 04/12/2023).

[68] ——, "J-Link RTT – Real Time Transfer," https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/, (Retrieved: 04/12/2023).

[69] ——, "J-Link SDK – Integrate J-Link Support into Applications," https://www.segger.com/products/debug-probes/j-link/technology/j-link-sdk/, (Retrieved: 04/12/2023).

[70] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC'12)*, 2012.

[71] B. Seri, G. Vishnepolsky, and D. Zusman, "Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS," https://www.armis.com/research/urgent11/, (Retrieved: 04/12/2023).

[72] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with Bit-Precision," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit

[73] M. Shen, J. C. Davis, and A. Machiry, "Towards automated identification of layering violations in embedded applications (wip)," in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2023, pp. 143–147.

[74] J. Shi, W. Li, W. Wang, and G. Le, "IPEA," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8296807

[75] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1275–1295.

[76] STMicroelectronics, "ClockAndWeather," https://github.com/STMicroelectronics/STM32CubeH7/tree/v1.11.0/Projects/STM32H7B3I-DK/Demonstrations/ClockAndWeather, (Retrieved: 04/12/2023).

[77] STMicroelectronics, "STM32Cube initialization code generator," https://www.st.com/en/development-tools/stm32cubemx.html, (Retrieved: 04/12/2023).

[78] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1433–1449.

[79] The Clang Team, "Hardware-assisted AddressSanitizer Design Documentation," https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html, (Retrieved: 04/12/2023).

[80] ——, "MemorySanitizer," https://clang.llvm.org/docs/MemorySanitizer.html, (Retrieved: 04/12/2023).

[81] ——, "ThreadSanitizer," https://clang.llvm.org/docs/ThreadSanitizer.html, (Retrieved: 04/12/2023).

[82] ——, "UndefinedBehaviorSanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, (Retrieved: 04/12/2023).

[83] The kernel development community, "The Kernel Address Sanitizer (KASAN)," https://www.kernel.org/doc/html/latest/dev-tools/kasan.html, (Retrieved: 04/12/2023).

[84] Vedere Labs, "OT:ICEFALL: 56 Vulnerabilities Caused by Insecure-by-Design Practices in OT," https://www.forescout.com/blog/ot-icefall-56-vulnerabilities-caused-by-insecure-by-design-practices-in-ot/, (Retrieved: 04/12/2023).

[85] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–36, 2021.

[86] S. H. Yong and S. Horwitz, "Protecting c programs from attacks via invalid pointer dereferences," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003, pp. 307–316.

[87] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*, San Diego, UNITED STATES, 02 2014.

[88] Zalewski, Michal, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/, (Retrieved: 04/12/2023).

[89] T. Zhang, D. Lee, and C. Jung, "Bogo: Buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 631–644.

[90] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen

[91] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

APPENDIX A
SUPPLEMENTARY APPENDIX

A. CVE Study

We list the classification of 37 CVEs in Table X.

B. Run-time Overhead

We list the run-time overhead of *IPEA-San* and ASan in Table XI. The upper part shows the results for BEEBS and the lower part shows the results for real-world applications

TABLE X. CVE CLASSIFICATION (NMC: NON-LINEAR MEMORY CORRUPTION, LMC: LINEAR MEMORY CORRUPTION, O: OTHERS, IO: INTRA-OBJECT BUFFER OVERFLOW).

Library	CVE	Type
FreeRTOS+TCP TCP/IP	CVE-2018-16522	O
	CVE-2018-16525	LMC
	CVE-2018-16526	NMC
	CVE-2018-16528	O
	CVE-2018-16523	O
	CVE-2018-16524	NMC
	CVE-2018-16527	NMC
	CVE-2018-16599	NMC
	CVE-2018-16600	NMC
	CVE-2018-16601	LMC
	CVE-2018-16602	NMC
	CVE-2018-16603	NMC
	CVE-2018-16598	O
Contiki-NG	CVE-2020-12140	LMC
	CVE-2020-12141	NMC
	CVE-2023-23609	LMC
	CVE-2023-31129	O
	CVE-2022-41873	NMC
	CVE-2022-41972	O
	CVE-2019-9183	LMC
	CVE-2023-28116	LMC
Zephyr	CVE-2021-3319	O
	CVE-2021-3320	O
	CVE-2021-3321	LMC
	CVE-2021-3322	O
	CVE-2021-3323	LMC
	CVE-2021-3330	LMC
	CVE-2020-10064	LMC
	CVE-2021-3329	O
	CVE-2022-3806	O
	CVE-2023-0359	O
NXP SDK	CVE-2021-38258	NMC
	CVE-2021-38260	NMC
STM32 SDK	CVE-2021-34259	NMC
	CVE-2021-34260	NMC
	CVE-2021-34262	O
	CVE-2021-42553	LMC & IO

TABLE XI. Normalized run-time overhead (the higher, the worse).

Firmware	**Baseline (ms)**	***IPEA-San* (x)**	**ASan (x)**
aha-compress	959	1.58	1.19
aha-mont64	942	1.55	1.03
bs	807	1.67	1.13
bubblesort	1,171	4.69	4.72
cnt	876	1.55	1.24
compress	890	1.67	1.27
cover	858	1.59	1.02
crc	841	1.64	1.18
crc32	1,258	1.30	1.25
ctl-stack	1,041	2.13	2.09
ctl-string	974	1.83	1.55
ctl-vector	1,123	1.84	1.80

Continued on next page

– continued from previous page

Firmware	**Baseline (ms)**	***IPEA-San* (x)**	**ASan (x)**
cubic	3,869	1.14	1.04
dijkstra	6,950	3.26	5.92
dtoa	1,119	1.75	1.18
duff	792	1.71	1.29
edn	1,168	3.28	4.97
expint	820	1.69	1.01
fac	846	1.56	1.01
fasta	2,506	1.00	4.92
fdct	881	1.64	1.13
fibcall	836	1.58	1.00
fir	2,319	8.49	6.31
frac	1,652	1.33	1.02
huffbench	3,281	4.36	8.15
insertsort	831	1.66	1.00
janne_complex	830	1.61	1.08
jfdctint	845	1.67	1.20
lcdnum	816	1.65	1.14
ludcmp	835	1.73	1.32
matmult-float	1,046	1.79	1.93
matmult-int	1,742	2.94	5.07
minver	847	1.67	1.24
nbody	18,433	1.19	1.32
ndes	1,495	1.89	3.01
nettle-aes	1,507	2.60	3.23
nettle-arcfour	901	2.25	2.25
nettle-cast128	1,044	1.74	1.39
nettle-des	965	1.76	1.42
nettle-md5	876	1.56	1.10
nettle-sha256	961	1.74	1.26
newlib-exp	854	1.54	1.06
newlib-log	829	1.63	1.10
newlib-mod	810	1.62	1.08
newlib-sqrt	829	1.64	1.10
ns	934	1.67	1.30
nsichneu	996	1.54	1.20
prime	936	1.50	1.17
qrduino	7,259	3.63	7.01
qsort	833	1.68	1.12
qurt	930	1.55	1.01
recursion	862	1.58	1.12
rijndael	8,569	3.68	6.18
select	826	1.63	1.10
sglib-arraybinsearch	886	1.73	1.37
sglib-arrayheapsort	927	2.20	2.39
sglib-arrayquicksort	907	1.88	1.81
sglib-dllist	1,085	3.94	3.39
sglib-hashtable	1,274	2.26	2.14
sglib-listinsertsort	1,184	2.97	4.08
sglib-listsort	996	3.40	2.89
sglib-queue	955	2.29	2.45

Continued on next page

– continued from previous page

Firmware	Baseline (ms)	*IPEA-San* (x)	ASan (x)
sglib-rbtree	1,763	3.08	4.20
slre	1,469	1.61	1.50
sqrt	6,784	1.07	1.36
st	3,247	1.12	1.13
statemate	899	1.61	1.25
stb_perlin	2,077	1.41	1.83
stringsearch1	897	1.90	1.70
strstr	860	1.53	1.11
tarai	836	1.60	1.08
template	838	1.56	1.01
ud	867	1.68	1.25
whetstone	13,266	1.06	1.11
Min	-	1.00	1.00
Max	-	8.49	8.15
Geo. Mean	-	1.86	1.67
PinLock	12	1.00	1.08
CNC	10,532	1.02	1.07
nRF52-Keyboard	3,702	1.02	1.05
ClockAndWeather	46,082	1.22	1.12
AudioPlayer	7,262	1.03	1.03
WeighScale	1,423	1.59	1.00
HttpServer	1,651	1.63	-
U-Disk	247	1.08	-
MQTT-Echo	3,672	1.32	-
App-Scheduling	1,320	1.01	1.01
App-Timers	4,189	1.02	1.01
App-IRQs	5,612	1.01	1.00
Min	-	1.00	1.00
Max	-	1.63	1.12
Geo. Mean	-	1.14	1.04

C. Bug Details

STM32-WiFi. The WiFi driver interacts with the WiFi module through AT commands via SPI interface. Assuming there is a malicious Access Point nearby, it can send crafted messages over the air, thereby influencing the AT command parser. The buffer overflow occurs in the function named `SPI_WIFI_ResetModule()` where the bounds checking against the array `Prompt` is missing, allowing attackers to corrupt the stack. We list the vulnerable code in Listing 3.

```
1  int8_t SPI_WIFI_ResetModule(void){
2    uint32_t tickstart = HAL_GetTick();
3    uint8_t Prompt[6];
4    uint8_t count = 0;
5    ...
6    while(WIFI_IS_CMDDATA_READY()){
7      Status = HAL_SPI_Receive(&hspi, &Prompt[count], 1, 0
       xFFFF);
8      count += 2; // bug: count is not checked in line 7
9      if(((HAL_GetTick()-tickstart) > 0xFFFF) || (Status!=
       HAL_OK)){
10       WIFI_DISABLE_NSS();
11       return -1;
12     }
13   }
14   ...
15 }
```

Listing 3. The buffer overflow bug in STM32 WiFi driver.

NXP-USB (CVE-2023-38749). The USB host driver, running on the MCU, needs to parse the descriptors provided by the USB device (e.g., a USB disk). If the USB device is malicious, it can send arbitrary descriptors, causing memory errors when the USB host parses them. In this bug, the function `USB_HostProcessCallback` processes the retrieved descriptor. It first extracts the descriptor length to `configureDesc->wTotalLength`. Then, the old descriptor `deviceInstance->configurationDesc` is freed at line 13. If everything is correct, a buffer for the new descriptor will be allocated (line 20). However, since `configureDesc->wTotalLength` can be controlled by the attacker, the function can return on error conditions at either line 16 or 18. Later, a dangling pointer to the already-freed descriptor is dereferenced at line 27, causing a UAF error.

```
1  static usb_status_t USB_HostProcessCallback(...) {
2    ...
3    usb_descriptor_configuration_t *configureDesc;
4    ...
5    usb_host_device_enumeration_status_t state;
6    ...
7    switch (state) {
8    case kStatus_DEV_GetCfg9:
9      ...
10     deviceInstance->configurationLen =
        USB_SHORT_FROM_LITTLE_ENDIAN_ADDRESS(configureDesc->
        wTotalLength);
11     if (deviceInstance->configurationDesc != NULL) {
12       ...
13       OSA_MemoryFree(deviceInstance->configurationDesc);
14       deviceInstance->configurationDesc = NULL;
15     }
16     if (deviceInstance->configurationLen < 9U)
17       return kStatus_USB_Error;
18     if (deviceInstance->configurationLen > MAX_LENGTH)
19       return kStatus_USB_Error;
20     ... // allocate a new descriptor
21   }
22 }
23 static usb_status_t USB_HostNotifyDevice(...)
24 {
25   ...
26   //bug: bInterfaceClass is a dangling pointer
27   if (((usb_descriptor_interface_t *)deviceInstance->
       configuration.interfaceList[interfaceIndex].
       interfaceDesc)->bInterfaceClass ==
       USB_HOST_HUB_CLASS_CODE)
28     ...
29   ...
30 }
```

Listing 4. The UAF bug in NXP USB driver (code slightly changed for easy presentation).

STM32-USB. This is a Denial-of-Service (DoS) bug triggered by malformed descriptor input. The non-responding state can only be recovered by manually resetting the MCU. This bug can be discovered without using *IPEA-San*. When parsing the descriptor, the function `USBH_GetNextDesc` moves the pointer `ptr` forward to the next descriptor. If the `bLength` field of current descriptor is 0, `ptr` will never move forward, leading to a dead loop in its caller.

```
1  USBH_DescHeader_t *USBH_GetNextDesc(uint8_t *pbuf,
       uint16_t *ptr)
2  {
3    USBH_DescHeader_t *pnext;
4    *ptr += ((USBH_DescHeader_t *)(void *)pbuf)->bLength;
5    pnext = ...;
6    return (pnext);
7  }
```

Listing 5. The DoS bug in STM32 USB driver.

SEGGER-USB. SEGGER emUSB is a proprietary IoT library. The bug detail is not disclosed due to the NDA protocol with SEGGER.

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The AEC-approved artifact can be found at https://doi.org/10.5281/zenodo.8296807. We also maintain the project on GitHub. For the updated versions, please refer to https://github.com/MCUSec/IPEA.

2) Hardware dependencies:

- Debugger
  - SEGGER J-Link (Pro/Edu/Edu Mini/Onboard)
- Development board
  - NXP FRDM-K64F

3) Software dependencies:

- Ubuntu 22.04 LTS x86_64
- J-Link Software and Documentation pack
- J-Link Runtime Library
- Arm GNU Toolchain
- LLVM 13
- Python 3.x
- Other libraries: *pyelftools*, *libjsoncpp*, *spdlog*, etc.

4) Benchmarks:

- Juliet C/C++ Testsuite
- BEEBS

B. Artifact Installation & Configuration

This section details the high-level installation and configuration steps to prepare the *IPEA* framework.

Install the J-Link Software and Documentation pack

- Download the J-Link Software and Documentation pack from the link: https://www.segger.com/downloads/jlink/JLink_Linux_V758e_x86_64.deb (accept the terms of use when prompted)
- Install in the command line:

```
$ sudo dpkg -i /path/to/JLink_Linux_V758e_x86
_64.deb
```

Install the J-Link Runtime Library

- Download the J-Link Runtime Library from the link: https://www.segger.com/downloads/jlink/JLink_Linux_V758e_x86_64.tgz (accept the terms of use when prompted)
- Extract *libjlinkarm.so.7.58.5* from the package and copy it to */usr/lib* directory:

```
$ sudo cp /path/to/libjlinkarm.so.7.58.5 /usr/lib
$ sudo ldconfig
$ sudo ln -s /usr/lib/libjlinkarm.so.7 \
             /usr/lib/libjlinkarm.so
```

Install the Arm GNU toolchain

- Download the latest version of the Arm GNU toolchain package from https://developer.arm.com/downloads/-/gnu-rm and unpack it.
- Add the toolchain path:

```
export ARMGCC_DIR=/path/to/arm-toolchain
export PATH=${ARMGCC_DIR}/bin:$PATH
```

Install LLVM-13 and other dependencies

- *LLVM-13* and other libraries:

```
$ sudo apt install llvm-13-dev clang-13 \
    cmake libjsoncpp-dev libconfig-dev libelf-dev
$ sudo ln -s /usr/bin/clang-13 /usr/bin/clang
$ sudo ln -s /usr/bin/clang++-13 /usr/bin/clang++
```

- Python dependencies: *pyelftools* and *cmsis-svd*:

```
$ pip install pyelftools cmsis-svd
```

- Install *spdlog*:

```
$ git clone https://github.com/gabime/spdlog.git
$ mkdir -p spdlog/build
$ cd spdlog/build && cmake ..
$ make && sudo make install
```

Build *IPEA* framework

- Clone the latest source code from https://github.com/MCUSec/IPEA to your working directory
- Set the `IPEA_HOME` environment variable to your working directory:

```
export IPEA_HOME=/path/to/IPEA_Source_Code
```

- Build the *IPEA* framework:

```
$ cd ${IPEA_HOME}
$ ./build.sh
```

- Add the following paths to the `PATH` environment variable:

```
export PATH=${IPEA_HOME}/build/AFL:
    ${IPEA_HOME}/build/unittest:
    ${IPEA_HOME}/scripts:$PATH
```

C. Major Claims

The following major claims are made:

- (C1): Besides the traditional memory safety bugs which can be also detected by ASan, *IPEA*-San can detect intra-object buffer overflow and peripheral-based buffer overflow. This is proven by experiment (E1), for which the results are illustrated in Table II.
- (C2): *IPEA*-San has lower FP and FN rates than ASan. This is proven by experiment (E2), for which the results are illustrated in Table IV.
- (C3): *IPEA*-San has lower flash and SRAM overhead but incurs slightly higher overhead in performance than ASan. This is proven by experiment (E3), for which the results are illustrated in Table V and Table XI.
- (C4): The combination of *IPEA*-Fuzz and *IPEA*-San is able to find memory bugs in MCU firmware. This is proven by experiment (E4), for which the results are illustrated in Table IX.

D. Evaluation

1) Experiment (E1): [2 human-minutes + 10 compute-minutes]: This experiment is to prove that *IPEA*-San is able to detect all eight kinds of memory bugs listed in Table II. By fuzzing the *Toy* firmware with *IPEA*-Fuzz, eight unique crashes should be found.

[Preparation]

- Connect SEGGER J-Link to the JTAG/SWD port of FRDM-K64F development board

[Execution]

- Build the *Toy* firmware then run *IPEA*-Fuzz with the following commands:

```
$ cd ${IPEA_HOME}/fw_samples/Toy
```

```
$ make ipea
$ run_afl.py -b ./toy -i ./fuzz_input -t 1000
```

- Press Ctrl+C to stop the fuzzing.

[Results]

- The fuzzing result can be found from the output directory (default is *output*).
- Use *ipea-unittest* tool to test a testcase:

```
$ cat output/crashes/<use_case_name> | \
   run_unittest.py -b toy -t 1000
```

- The execution results will be saved in *tracelog_0.txt*. If a crash is detected, the call stack information will be saved in a file *callstack.txt*.

2) Experiment (E2): [10 human-minutes + 2 compute-hours]: This experiment is to evaluate the correctness of *IPEA-San* in detecting memory safety bugs in the Juliet C/C++ Testsuite (Juliet).

[Preparation]

- Connect SEGGER J-Link to the JTAG/SWD port of FRDM-K64F development board.

[Execution]

- Run Juliet with *IPEA*-San:

```
$ cd ${IPEA_HOME}/projects/MCU_Juliet_Testsuite
$ run_juliet.py -p . -c mk64f.conf
```

- Run Juliet with ASan:

```
$ run_juliet.py -p . -c mk64f.conf --use-asan
```

[Results]

- The results (i.e., numbers of FPs and FNs in each CWE) will be saved in two files *report_ipea.json* and *report_asan.json* for *IPEA*-San and ASan, respectively.

3) Experiment (E3): [5 human-minutes + 30 compute-hours]: This experiment is to evaluate the performance overhead of *IPEA*-San on the BEEBS benchmark.

[Preparation]

- Connect SEGGER J-Link to the JTAG/SWD port of FRDM-K64F development board.

[Execution]

- Run the baseline BEEBS without any instrumentation:

```
$ cd ${IPEA_HOME}/projects/BEEBS
$ run_beebs.py -p . -c mk64f.conf
```

- Run BEEBS with *IPEA*-San:

```
$ run_beebs.py -p . -c mk64f.conf -s ipea
```

- Run BEEBS with ASan:

```
$ run_beebs.py -p . -c mk64f.conf -s asan
```

[Results]

- The results (time consumption of each program) will be saved in three files *report_none.json*, *report_ipea.json* and *report_asan.json* for baseline, *IPEA*-San and ASan, respectively.
- Performance overhead of each sanitizer can be obtained by comparing the corresponding time consumption with the baseline.

4) Experiment (E4): [5 human-minutes + 24 compute-hours]: This experiment is to evaluate the capability of finding bugs in real-world MCU firmware when combining *IPEA*-San with *IPEA*-Fuzz.

[Preparation]

- Connect SEGGER J-Link to the JTAG/SWD port of FRDM-K64F development board.
- Plug a USB drive to the J22 connector via a micro USB OTG cable.

[Execution]

- Run *IPEA*-Fuzz through *run_afl.py*:

```
$ cd ${IPEA_HOME}/fw_samples/USB-Host
$ make ipea
$ run_afl.py -b usb_host -i ./fuzz_input -t 3000
```

[Results]

- A use-after-free bug would be found within a couple of hours.
- The fuzzing results can be found from the *output* directory. Test the crashing testcases through *run_unittest.py* as described in E1.