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], [55], [40], [22], [46], [57], [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 framework 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-
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 [83]. 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 compile using ASan but succeeded using IPEA-San. 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 [60, 61, 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.

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] observes 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., GDBSover) 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.

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-collceting 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.
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-C), 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.

Listing 1. The basic idea of IPEA-San.

```c
1 p1 = malloc(size);
2 //collect bounds info for heap objects
3 send_to_PC(OP_NEW, p1, p1 + size, p1_id);
4 p2 = malloc(size);
5 //propagate to temporary pointers
6 send_to_PC(OP_PROP, p2_id, p1_id);
7 g_p = p1;  //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_id, sizeof(p2), p2_id);
13 value = p2->a;
```

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.

```
#define __IOM volatile
typedef struct {
  // Offset: 0x000 (R/W) Interrupt Set Enable Register
  __IOM uint32_t ISER[8U];
  uint32_t RESERVED0[24U];
  // Offset: 0x300 (R/W) Interrupt Priority Register
  __IOM uint8_t IP[240U];
  uint32_t RESERVED5[644U];
  NVIC_Type
  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 above.

**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 [53]. The pointer to a sub-object is typically derived by the address-of operator. For example, in the statement `g = a (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, offset_of(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.

**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 sub-sub-object) corresponding to different pointers that can access it. As shown in the object A has a sub-object A.X which in turn has a sub-sub-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-
agination 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 invariants 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 deallocation 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. 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 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 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.

```c
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.

<table>
<thead>
<tr>
<th>Memory Errors</th>
<th>IPEA-San</th>
<th>ASan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack-based Buffer Overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heap-based Buffer Overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Global-based Buffer Overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use-after-free</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Double Free</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Null Pointer Dereference</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Peripheral-based Buffer Overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intra-object Overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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-NucleoF411R (STM32F4, 512KB flash/128KB SRAM), STM32H7B31-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 have 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 [33]). 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 error [57], 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.

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-
construction 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 randomness (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 [11] (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 re-

views 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 RTOs (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.
- CNN is based on the GRBL-Advanced project [7] that implements a CNN (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 verifies its SHA-256 hash value. It runs Candle 2 [65] to control the stepping motors.
- HttpServer implements an HTTP server on top of lwIP TCP/IP stack [41].
- 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
projects based on FreeRTOS for the popular Raspberry Pi RP2040 MCUs.

TABLE VI. TESTED REAL-WORLD APPLICATIONS

<table>
<thead>
<tr>
<th>Application</th>
<th>Target MCU</th>
<th>OS</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>PinLock</td>
<td>K64F</td>
<td>Bare-metal</td>
<td>[29]</td>
</tr>
<tr>
<td>CNC</td>
<td>STM32F4</td>
<td>Bare-metal</td>
<td>[91]</td>
</tr>
<tr>
<td>nRF52-Keyboard</td>
<td>nRF52</td>
<td>Bare-metal</td>
<td>[92]</td>
</tr>
<tr>
<td>ClockAndWeather</td>
<td>STM32H7</td>
<td>FreeRTOS</td>
<td>[93]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>K66F</td>
<td>Bare-metal</td>
<td>NXP SDK [60]</td>
</tr>
<tr>
<td>WeighScale</td>
<td>K66F</td>
<td>Bare-metal</td>
<td>NXP SDK [60]</td>
</tr>
<tr>
<td>HtpServer</td>
<td>K66F</td>
<td>Bare-metal</td>
<td>NXP SDK [60]</td>
</tr>
<tr>
<td>U-Desk</td>
<td>K64F</td>
<td>FreeRTOS</td>
<td>NXP SDK [60]</td>
</tr>
<tr>
<td>MQTT-Echo</td>
<td>K64F</td>
<td>FreeRTOS</td>
<td>NXP SDK [60]</td>
</tr>
<tr>
<td>App-Scheduling</td>
<td>RP2040</td>
<td>FreeRTOS</td>
<td>[12]</td>
</tr>
<tr>
<td>App-Timers</td>
<td>RP2040</td>
<td>FreeRTOS</td>
<td>[12]</td>
</tr>
<tr>
<td>App-IRQs</td>
<td>RP2040</td>
<td>FreeRTOS</td>
<td>[12]</td>
</tr>
</tbody>
</table>

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 Table VIII A. In Table VII, 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 MCUs chips. We dumped the configuration information of the first 500 MCU chips returned from the DigiKey website. 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.

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 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 MCUs chips. We dumped the configuration information of the first 500 MCU chips returned from the DigiKey website. 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.

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 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 MCUs chips. We dumped the configuration information of the first 500 MCU chips returned from the DigiKey website. 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.

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 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.
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 §VIA-A), 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).

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 drove 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...
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 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 8A-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], [54], [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 test case 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.

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. 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 [83], [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

3https://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 [57], [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], [89] 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


APPENDIX A
SUPPLEMENTARY APPENDIX

A. CVE Study

We list the classification of 37 CVEs in Table XI.

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).

<table>
<thead>
<tr>
<th>Library</th>
<th>CVE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeRTOS+TCP  TCP/IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16522</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16525</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16526</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16528</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16523</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16524</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16599</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16600</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16601</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16602</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16603</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2018-16598</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Contiki-NG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2020-12140</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2020-12141</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2023-23609</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2023-31129</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2022-41783</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2022-41972</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2019-9183</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2023-28116</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>Zephyr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3319</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3320</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3321</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3322</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3323</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3330</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2020-10064</td>
<td>LMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3329</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2022-3806</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-3359</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>NXP SDK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2021-38258</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-38260</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>STM32 SDK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2021-34259</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-34260</td>
<td>NMC</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-34262</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>CVE-2021-42553</td>
<td>LMC &amp; IO</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Continued on next page
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 bug.

### 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 is missing, allowing attackers to corrupt the stack. We list the vulnerable code in **Listing 3**.

```c
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);
8    count += 2; // bug: count is not checked in line 7
9    if(((HAL_GetTick() - tickstart) > 0x0F0F) || (Status==HAL_OK)) {
10      WIFI_DISABLE_NSS();
11      return -1;
12    }
13  }
14  return 0;
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

### 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.

```c
1 static usb_status_t 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);
```

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

### 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
     o SEGGER J-Link (Pro/Edu/Edu Mini/Onboard)
   - Development board
     o 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

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

- $ 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++
- $ sudo ln -s /usr/bin/llvm-config /usr/bin/llvm-config
- $ sudo apt install pyelftools and cmsis-svd
- $ pip install pyelftools cmsis-svd
- $ 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
  $ export PATH=${IPEA_HOME}/build:AFL:scripts
  $ export PATH=${IPEA_HOME}/build/unittest:

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.
- Plug a USB drive to the J22 connector via a micro USB OTG cable.

[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}/projects/USB-Host
  $ make ipea
  $ run_afl.py -b usbHost -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.