# EmbedWatch: Fat Pointer Solution for Detecting Spatial Memory Errors in Embedded Systems

Davide Rusconi
davide.rusconi@unimi.it
University of Milan
Milan, Italy

Matteo Zoia
matteo.zoia@unimi.it
University of Milan
Milan, Italy

Luca Buccioli
luca.buccioli@unimi.it
University of Milan
Milan, Italy

Fabio Pierazzi
fabio.pierazzi@kcl.ac.uk
King's College London
London, United Kingdom

Danilo Bruschi
danilo.bruschi@unimi.it
University of Milan
Milan, Italy

Lorenzo Cavallaro
l.cavallaro@ucl.ac.uk
University College London
London, United Kingdom

Flavio Toffalini
flavio.toffalini@epfl.ch
EPFL
Lausanne, Switzerland

Andrea Lanzi
andrea.lanzi@unimi.it
University of Milan
Milan, Italy

## ABSTRACT

This paper introduces EmbedWatch, an innovative crash reporting system specifically designed for embedded devices. EmbedWatch integrates fat pointer principles with remote attestation, efficiently addressing spatial memory errors across various memory segments, including stack, heap, and global variables. The system's notable feature is its vulnerability analysis capability, which precisely pinpoints the exact code segment responsible for an error, significantly enhancing error detection and resolution accuracy. We evaluate the effectiveness and practicality of EmbedWatch by evaluating real-world firmware and CWEs. We show that EmbedWatch accurately detects the vulnerability analysis of spatial memory errors in the analyzed firmware with a negligible overhead RANGE (0.01% - 2.33%), GEOMETRIC MEAN 0.228 +0.4% ENCRYPTION PROTOCOL .

## CCS CONCEPTS

• **Security and privacy** → *Information flow control*; Trusted computing; • **Computer systems organization** → **Embedded systems**.

## 1 INTRODUCTION

Embedded systems have unique characteristics such as limited resources, restrictions on updates, and real-time constraints, which complicate the application of traditional attack detection methods. In particular, IoT firmware may suffer from memory errors that lead to data-only and control-flow attacks [33] which can have severe consequences such as crashes, data breaches, or even physical damage. Previous efforts to design detection systems to address memory errors in the embedded systems landscape have typically relied on remote attestation methods, incorporating techniques such as Control-Flow Integrity (CFI) [13, 39, 45]. However, these techniques lack their ability to perform *vulnerability analysis*, which is essential to identify vulnerabilities and prevent future intrusions. Crash reporting (e.g., vulnerability analysis) is crucial yet challenging in embedded systems due to the necessity of identifying and addressing vulnerabilities in a constrained environment where traditional debugging tools and methodologies are often limited or impractical. In contrast, in less constrained environments, such as desktop PCs, many products routinely collect and report crash data to vendors. This practice, exemplified by Mozilla and Ubuntu with their crash reporting systems [2, 8], aims to improve the quality and security of their products.

Building upon the goal of designing a crash reporting system for embedded devices, this paper presents EmbedWatch, *- a groundbreaking system that merges fat pointer principles with remote attestation, tailored specifically for embedded devices.* EmbedWatch *distinguishes itself by its ability to effectively enforce the spatial memory property, performing a comprehensive analysis of various memory segments including stack, heap, and global variables.* The key innovation of EmbedWatch lies in its vulnerability analysis capability, which accurately identifies the exact code segment responsible for a bug, enhancing precision in error detection and resolution. Unlike conventional approaches that are reliant on Control Flow Integrity (CFI) systems [13, 39, 45], EmbedWatch introduces a broader and more comprehensive strategy. This system extends beyond merely detecting control-flow hijacking vulnerabilities; it adeptly identifies and locates vulnerabilities associated with control and data attacks. Consequently, EmbedWatch equips IoT devices with the capability to autonomously generate and send detailed crash reports of such attacks, improving their security and response mechanisms.

Our framework presents an innovative static framework called a data graph model that utilizes node selection and runtime data to identify Out-of-Bound (OOB) memory errors, including memory structure data attacks. *It is important to note that we exclude spatial memory errors that do not lead to successful attacks, such as scenarios where the overflow cannot be manipulated by the input program.* This model not only pinpoints critical locations in the code prone to OOB errors, but also guarantees their thorough monitoring during execution. The data graph model merges flawlessly with the existing SVF [38] framework, boosting its capabilities for accurate interprocedural analysis and firmware code instrumentation. The instrumented firmware is specifically configured to monitor memory structures that are more susceptible to attacks, such as buffers, thus effectively minimizing run-time overhead. To further optimize performance and reduce this overhead, we have implemented additional design optimizations based on the cache concept that enhance efficiency without compromising security. During runtime, with the support of a trusted anchor, such as ARM TrustZone [25], the device securely transmits a minimal, yet highly reliable, data trace that represents the current running status of the program represented by a data graph model. This approach not only secures the data in transit, but also ensures that the information is reflective of the real-time execution state of the firmware. Upon reaching the IoT backend, these data undergo a thorough processing phase. Here, the system analyzes the information to detect any instances of spatial memory violations. The backend is equipped with an algorithm capable of dissecting and understanding the details of these violations. As a result, the IoT backend generates detailed crash reports that pinpoint the vulnerability's origin in the source code.

EmbedWatch has been developed to enable firmware vendors to produce software that communicates the execution status in real time, helping to quickly deploy patches. Consequently, we argue that it is crucial to have access to the firmware's source code. This access reveals detailed information, including the position of the memory structures, the precise sizes of each memory object, and the techniques employed in memory management allocation.

To assess EmbedWatch, we conducted a set of experiments that measure runtime overhead and stress the security guarantees. Specifically, we deploy EmbedWatch in real-world firmware containing genuine CWEs [17]. Our results show that EmbedWatch exhibits remarkable precision in identifying the underlying source of memory errors within the analyzed firmware. Additionally, we observe a runtime overhead of range (0.01% - 2.33%), geometric mean 0.228 +0.4% encryption protocol .

In summary, this paper makes the following contributions.

- We present EmbedWatch, a groundbreaking crash reporting system that employs a fat pointer technique for embedded devices in production environments. We have created, implemented, and successfully tested EmbedWatch, which accurately detects vulnerabilities in spatial memory attacks, thus enhancing IoT backends (§6). The source code for EmbedWatch is open source and can be accessed at [4].
- We introduce a static framework named the data graph model (§6), which examines firmware code by identifying nodes and runtime data that can detect Out-of-Bound (OOB) memory errors. This data graph model integrates effortlessly with the current SVF framework [38], improving its precision in interprocedural analysis. Moreover, our framework incorporates optimization techniques to minimize performance overhead and to address data attacks on memory structure fields.
- We evaluated EmbedWatch on real-world embedded programs that cover broad scenarios of use of IoT devices, demonstrating the efficacy and practicality of EmbedWatch in real-world cases. Our experimental results with in place optimizations demonstrate that EmbedWatch reduces the initial overhead from range (0.07% - 36.04%), geometric mean 1.4 +0.4% encryption protocol to range (0.01% - 2.33%), geometric mean 0.228 +0.4% encryption protocol

## 2 RELATED WORK

*Fat Pointer System.* Fat pointer systems, like Safe-C [16], CCured [31], and Cyclone [26], combine pointer values with associated bounds metadata into a single structure, differing in how they store this metadata. An alternate approach to managing bounds metadata is through shadow memory, used in memory safety systems like SoftBound [30], PAriCheck [46], Baggy Bounds Checking [15] and Intel MPX. This method links objects or pointers in the main memory with metadata stored in a separate shadow memory, creating a relationship between them. Recent developments have introduced "low-fat pointers", particularly effective in 64-bit systems with extensive pointer bit-width [23]. This innovative concept embeds bounds metadata directly within the pointer's representation, contrasting with fat pointer or shadow space methods that store metadata in a separate location. Our system functions similarly to a fat pointer system. Implementing fat pointer technique in embedded systems encounters several challenges. The main issue is that the instrumentation and runtime checking of pointers required by these systems can significantly slow down execution, making them inefficient for embedded systems, which often have limited resources. To address these challenges, EmbedWatch has been designed to integrate a fat-pointer mechanism through source code instrumentation. However, unlike traditional methods that conduct bound checking directly on the device, EmbedWatch offloads this task to a remote verifier through a remote attestation system.

*Runtime Remote Attestation Attacks.* C-FLAT [13] is the first remote attestation protocol designed to verify the runtime correctness of embedded devices. It focuses on measuring control flow integrity and targets control flow attacks. This protocol has inspired future runtime remote attestation protocols, such as LO-FAT [22], which is a hardware implementation of C-FLAT that improves performance. ATRIUM [47] extends the protection offered by both C-FLAT and LO-FAT against physical attacks, that is, adversaries who physically tamper with the embedded device. After C-FLAT, new protocols were developed to address data-only attacks, such as LiteHax [21]. LiteHax measures firmware data-flow and leverages symbolic execution for verification, but it has a prohibitive overhead that makes it unsuitable for production environments. DIAT [14] proposes a coarse-grained data flow mechanism to verify the correct data transmission among the firmware modules and pairs it with a lightweight control flow attestation.

Among the notable works in the field of remote runtime attestation, OAT [39] and a recent refinement BLAST [45] emerges as the latest advancement, encompassing control flow verification and data-only verification for ARM-based embedded devices. However, these systems primarily focus on attack detection, identifying anomalies consistent with their respective attacker models. Regrettably, all these systems fail to provide crucial information on the location of vulnerabilities, leaving analysts and developers without the necessary insight to develop effective patches.

Another system that recently surfaced is IPEA [36] a framework that includes a sanitizer IPEA-san and a fuzzer IPEA-fuzz. Even if IPEA uses a remote-attestation-based architecture like EmbedWatch, the proposed systems target different use cases. Although IPEA facilitates the test process for embedded devices, EmbedWatch focuses on detecting spatial memory safety violations in deployed devices. This difference reflects in the running time overhead as IPEA incurs a significantly higher overhead compared to EmbedWatch, as expected from a sanitizer in both general-purpose and embedded contexts. Moreover EmbedWatch incorporates self-protection mechanism against potential exploits and a robust pre-processing analysis which is crucial to offloads heavy computational tasks from the verifier, making it a lightweight component in live detection use cases.

While exploring the realm of modern runtime remote attestation, several solutions have emerged, catering to software complexities beyond simple embedded firmware. Notably, ScaRR [41] and ReCFA [48] have paved the way for applying runtime remote attestation to services within cloud environments. However, their focus lies solely on monitoring control-flow integrity, disregarding data-flow aspects. On a different front, SgxMonitor [42] presents a pioneering runtime remote attestation protocol explicitly tailored for SGX enclaves. Although SgxMonitor introduces a stateful model, its approach remains confined to globally defined structures. It should be noted that these protocols, similar to those for embedded devices, do not perform vulnerability analysis, which is the primary objective of EmbedWatch.

*Vulnerabilities Analysis.* While exploring techniques to identify the location of bugs in the presence of anomalies, such as crashes, there are several options. Notably, sanitizers like AddressSanitizer [35] (ASan), or more recent optimized versions [49] (ASan--), stand out as state-of-the-art tools for monitoring spatial and temporal errors. However, it is important to note that sanitizers are primarily designed for testing purposes, rendering them unsuitable for production environments. They introduce significant performance and memory overhead while lacking protection against evasion techniques or attacks targeting the sanitizer itself. Alternatively, postmortem techniques analyze a core dump of the process to locate errors, bypassing the need for sanitizers. Examples include works such as [20, 43, 44], which may require additional contextual information like process runtime traces from Intel PT, as seen in REPT [20]. *Unlike sanitizers and post-mortem techniques,* EmbedWatch *excels in embedded production environments, as it avoids prohibitive overhead and is designed to withstand attacks.*

Zhenyu et al. introduced Ninja [32], a novel malware sandbox technique built upon ARM TrustZone. The primary objective of Ninja is to improve malware analysis by leveraging ARM TrustZone to effectively counter malware sandbox evasion. It is important to note that Ninja and EmbedWatch differ in their respective goals and constraints: (1) Ninja focuses on bolstering malware analysis capabilities, while EmbedWatch is specifically designed for deployment in production environments.(2) Ninja employs distinct methods to gather runtime information, such as capturing API/syscall invocations, whereas EmbedWatch targets bare-metal embedded devices and collects fine-grain memory access data, among other runtime information.

*Value Set Analysis.* In a specific domain of postmortem analysis, researchers have utilized Value Set Analysis [19] (VSA) to analyze the core dumps. VSA is a static analysis technique that infers variable ranges, aiding in error localization. To enhance the precision of VSA, POMP++ [29] proposes a VSA approach based on reverse execution from the crash point. On the other hand, DEEPVSA [24] addresses aliasing issues using a deep learning model. However, both POMP++ and DEEPVSA rely on specific hardware features such as Intel PT and rely on core dumps for analysis. Consequently, they are unsuitable for deployment in embedded systems.

## 3 BACKGROUND

In this section, we introduce background information on IoT attacks (§3.1) and ARM TrustZone (§3.2).

### 3.1 IoT Attacks

Advanced attacks, such as return-oriented programming (ROP) and data-only attacks, have also emerged as a threat to embedded devices, particularly those using the ARM architecture [28]. Given the poor security of these devices, it is advisable for IoT backends to assume that IoT devices in the field may be compromised and should not be fully trusted. Even though tools [13, 39], that identify memory attacks can be helpful, they often lack the ability to identify the root cause of vulnerabilities. This is a crucial limitation, as understanding the underlying cause of a vulnerability is essential to effectively address the issue.

### 3.2 ARM TrustZone

Our system utilizes ARM TrustZone technology to establish a Trusted Computing Base (TCB) [25]. TrustZone is a hardware feature that is available on both Cortex-A processors, used in mobile and high-end IoT devices, and Cortex-M processors, used in low-cost embedded systems. ARM TrustZone is a security extension to the ARM processor architecture that enables the creation of a secure environment to execute sensitive operations. It allows for the separation of a device's normal and secure worlds, where the normal world contains untrusted applications, and the secure world contains trusted applications and a small operating system.

The secure world is an isolated environment with specialized caches, banked registers, and private memory, which ensures that trusted applications and data are protected. TrustZone also enables the provision of per-device private keys and certificates, which facilitates secure communication and authentication. The normal world can access the secure world through a set of secure monitor calls (SMCs) which allows the normal world to request sensitive services from the secure world, such as signing or securely storing data. TrustZone is widely used in mobile devices, secure elements, IoT devices, and embedded systems to provide a secure environment for sensitive operations.

## 4 THREAT MODEL

Our system trusts the code running inside the Secure World (*e.g.,* the measurement engine) and assumes that attackers cannot bypass TrustZone protection. We also trust our compiler and the trampoline code, and assume that attackers cannot inject code into the Normal World or tamper with the instrumented code or the trampoline library. This can be enforced using code integrity protection methods for embedded devices [18, 27], which are not the focus of this paper. In the Normal World, attackers may exploit spatial memory error vulnerabilities [40] in embedded systems to launch various types of memory error attacks, such as Return-Oriented Programming (ROP) and Data-Oriented Programming (DOP) attacks. These attacks can compromise the control flow of the firmware application and access critical data from an embedded program. Additionally, attackers may abuse unprotected interfaces of the embedded program to force the device to perform unauthorized or unintentional operations. To address these issues, our system is designed to detect these types of attacks against the system (§7.3). Generally, our system adheres to the defensive strategies employed by fat-pointer systems.

## 5 EMBEDWATCH **OVERVIEW**

Our system offers a variety of benefits compared to classic vulnerability analyzers [29, 43, 44], such as (1) eliminating the requirement of large and unnecessary core dumps, (2) reducing the amount of traced information created by the instruction execution trace [7], and (3) increasing resistance against attempts to circumvent the system by attackers. To provide an understanding of EmbedWatch, we present a practical example of vulnerability analysis in embedded systems (§5.1). Following that, we describe the system architecture in detail (§5.2).

### 5.1 Running Example

In Listing 1, we present the code snippet of the *Heat Press* firmware, extracted from our dataset, designed to control a press machine. The CWE vulnerability in question pertains to the outbound type and is specifically found within the Modbus library used for communication with peripheral devices.

In particular, the function `Modbus_poll` utilizes the `fgetc` function to read characters from the standard input (`stdin`) (Line 21). These characters are then stored in a buffer named `au8regs`. The firmware reads exactly one hundred characters and subsequently calls the `Modbus_get_FC3` function (Line 13), passing the `regs` buffer as an argument. The issue arises from the fact that the `Modbus_get_FC3` function lacks proper boundary checks. Consequently, if an attacker overflows the size of the `regs` buffer, adjacent memory locations can become corrupted, leading to potential security vulnerabilities.

To effectively identify out-of-bounds (OOB) vulnerabilities and provide meaningful crash reports, an analyst requires ample contextual information. This includes details such as (1) the specific object involved in the memory operation (e.g. memory structures, buffers), (2) the boundaries of the object, and (3) the stack trace at the time of the OOB occurrence.

Looking back at the example in Listing 1, we notice that the function `Modbus_get_FC3` operates on a buffer that could potentially be a parameter of the function call itself. The challenge here is that during the static analysis stage, we cannot determine which execution path the program will take, and consequently, we do not know which function parameters will be considered in case of multiple calls of the same function with different buffers. This lack of information during static analysis could lead to ambiguity in identifying the correct object involved in the out-of-bound (OOB) issue. However, EmbedWatch effectively resolves this ambiguity by generating a reliable trace that accurately tracks the progression of key variables in terms of their liveness. To achieve this, our instrumentation system records the base address of the relevant object, such as `au16regs` in the example (line 3), by leveraging run-time information, such as definition and the executed calls. By doing so, EmbedWatch provides the necessary context to identify the function's parameters and accurately pinpoint the location of the OOB vulnerability. For instance, when `Modbus_get_FC3` is called, our instrumentation system identifies that `au16regs` is being used as the `regs` parameter for `Modbus_get_FC3`. This crucial information reveals the actual object under consideration and allows us to infer the boundary of `regs` (i.e., based on the base address and size of `au16regs`). By understanding the reference of the variable `regs` and the addresses present in the loop, a remote agent can determine whether the firmware works within the correct limits. Specifically, it can verify whether the value of `regs[i]` resides within the memory range defined by `au16regs`.

Upon detecting an out-of-bound (OOB) issue, a remote agent possesses the capability to generate a crash report. This comprehensive report includes details on the location of the OOB (line 13) and the object involved (e.g. `au16re.g.s`, as defined in line 3). By tracing the sequence of function calls, particularly in our example, the `Modbus_get_FC3` and `Modbus_Poll` calls, in terms of the stack frame, the remote agent can effectively track the sequence of events leading up to the point of vulnerability. Within this context, three crucial challenges must be addressed:

- **(C1)** A technique consists of detecting erroneous memory accesses through the use of the memory state (§6.1).
- **(C2)** A remote attestation technique for remotely obtaining a concise representation of the memory state of the embedded device (§6.2, §6.3, and §6.4).
- **(C3)** A vulnerability analysis utilizes the inferred memory state and out-of-bound access to determine the bug's location (§6.5).

### 5.2 System Architecture

Figure 1 depicts the high-level architecture of EmbedWatch, which consists of two main components: *Offline Preparation* and *Online Verification*. In the *Offline Preparation* phase, we conduct program analysis techniques on the source code (❶) and instrument the firmware to trace runtime information (❷). Subsequently, we link the instrumented firmware with a trampoline library to facilitate communication with the trust application (❸), and install it in the embedded device (❹). The compilation process also generates a graph model M (❺), outlining the usage of critical buffers in the program. This model allows a remote agent to reconstruct the firmware's memory state, which aids in the effective detection of Out-of-Bound errors and vulnerability analysis.

During the *Online Verification* phase, the device communicates with the TrustZone (❻), transmitting a runtime trace T to the Verifier (❼). Finally, the remote Verifier uses the model M to identify potential outbound accesses and generates root cause reports based on trace T (❽). In the following section, we provide an overview of the software components and algorithms utilized by our system.

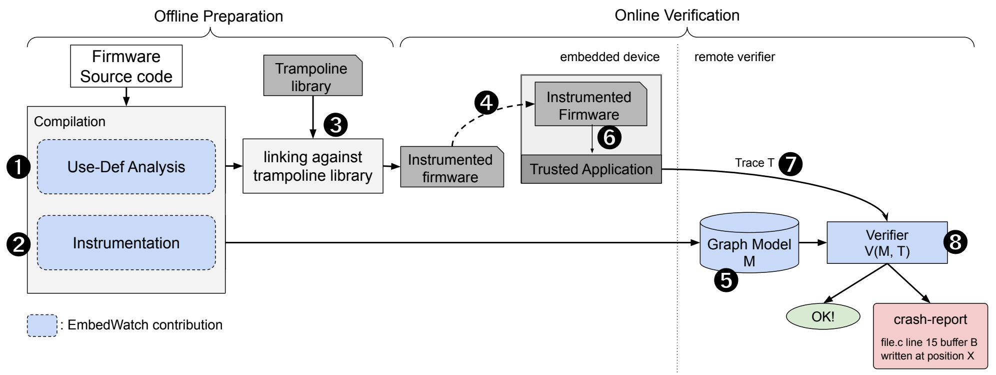


**Figure 1: EmbedWatch Architecture. During Off-line Preparation, the firmware source code is compiled and analyzed. In this phase, we obtain an instrumented firmware and a model that describe the data-flow relationship between inputs and internal buffers. During the online verification, the instrumented firmware is executed on a embedded device and reports a trace T to a Verifier. The latter validates the trace T against the model M. In case of violation, the Verifier reports the incident**

```
#include <stdio>
uint8_t au8Buffer[100];
uint16_t au16regs[20];

void Modbus_get_FC3(uint16_t  *regs)
{
    uint8_t u8byte, i;
    u8byte = 3;

    for (i=0; i< au8Buffer[ 2 ] /2; i++) {
        regs[ i ] = word(
          au8Buffer[ u8byte ],
          au8Buffer[ u8byte +1 ]);
        u8byte += 2;
    }
}

void Modbus_poll(){

  for(int i = 0; i < 100; i++)
    au8Buffer[i] = fgetc(stdin);

  Modbus_get_FC3(au16regs);
  u8state = 1;
}
```

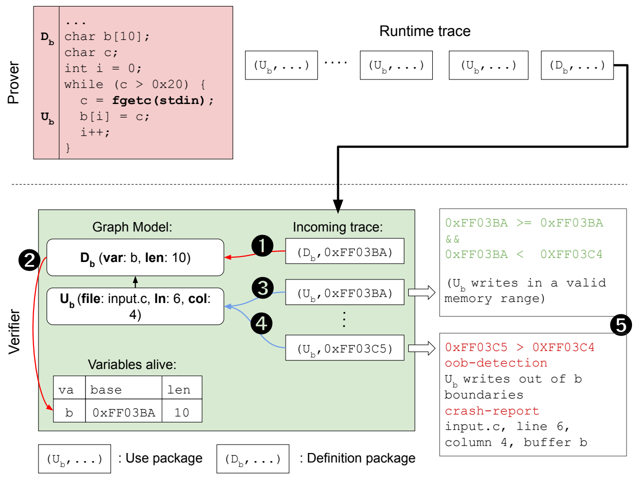
**Listing 1: Code snippet showcasing a CWE vulnerability in the Heat Press firmware, where the Modbus library is susceptible to out-of-bound access.**

## 6 EMBEDWATCH PROTOCOL

EmbedWatch is the first work that merges fat pointer principles with remote attestation, specifically tailored for embedded devices. As such, one of the major challenges is the ability to keep overhead low while effectively deducing the program variables utilized during the firmware's operational phase. To address this, our system monitors dynamic information such as: the objects that are active, determines their boundaries, and reconstructs the call stack when an out-of-bounds (OOB) error occurs. In the following sections, we detail how EmbedWatch achieves this goal by showing the protocol in §6.1. We expand the example by introducing the graph data modeling in §6.2, which allows us to find crucial instrumentation points. Then, we describe how the firmware collects runtime information in §6.3 and an optimization to reduce overhead in loops in §6.4. Finally, we show how EmbedWatch correctly detects and reports OOB errors in §6.5.

### 6.1 The Protocol in a Glance

EmbedWatch implements a remote attestation protocol between two points: the Prover, an inline agent defined within the firmware, and the Verifier, an external application aimed at identifying memory violations. To trace run-time information in the Prover, EmbedWatch uses instrumentation points provided by our data graph model. Without loss of generality, the protocol considers two main types of packets: *Definition*, which indicates the instantiation of new variables (e.g., alive variables) and *Use*, which represents a memory operation. During compilation time, our graph model assigns a unique ID for each *Definition* and *Use* of the variable. This ID is used to uniquely identify the information of the variables (e.g., memory boundaries) during the execution path and locate their corresponding definitions in a static view of the firmware. Figure 2 illustrates a simplified version of the protocol that only considers a single definition and the use of the same variable. The *Definition* $D_b$ is related with the variable b, whose size is 10 bytes (❶). The size information is retrieved by the static components of our framework and combined with the runtime base address from the incoming *Definition* and stored in the variables alive table (❷). At this point, the Verifier is aware that the buffer b is instantiated at the address 0xFF03BA. Then, for each *Use*, the Verifier inspects the model to infer the involved buffer and validate the memory access. In Figure 2, the *Use* $U_b$ is the only use connected to *Definition* $D_b$. Therefore, the Verifier infers that $U_b$ operates on the variable $b$. This allows the Verifier to decide whether the memory access falls within the variable range (❸). In case *Use* accesses outside the buffer, an out-of-bound is detected (❹). The operation *Use* reveals the location outside the bound in the source code, while the variable involved provides additional contextual information about the location of the code bug location (❺). The complete model is detailed in §6.2. At runtime, our protocol collects additional information to handle more complex cases such as aliasing and `struct` fields.



```
...
D_b  char b[10];
     char c;
     int i = 0;
     while (c > 0x20) {
       c = fgetc(stdin);
U_b    b[i] = c;
       i++;
     }
```

```
0xFF03BA >= 0xFF03BA
&&
0xFF03BA <  0XFF03C4

(U_b writes in a valid
memory range)
```

```
0xFF03C5 > 0XFF03C4
oob-detection
U_b writes out of b
boundaries
crash-report
input.c, line 6,
column 4, buffer b
```

**Figure 2: EMBEDWATCH Protocol. The prover sends a stream of Use and Definitons that represents the variable alive and their use. The Verifier infers the firmware state from the runtime trace and the Use-Def graph. The Use-Def graph reduce the size of the runtime trace in two ways (1) it avoids transmitting static information (*e.g.,* the buffer `b` length), and (2) it minimizes the instrumentation points (the only possible out-of-bound use is $\mathtt{U_b}$).**

## 6.2 Graph Data Modeling

The core mechanism of EMBEDWATCH revolves around a data graph model that represents connections between variables and their use in the firmware code, represented by the Graph Model M (❺ in Figure 1). This model represents our contribution and is utilized for the detection of Out-of-Bound errors and vulnerability analysis, enabling the Verifier to trace the lifetime of variables and determine their boundaries. Our primary goal is to perform a use-def analysis on the memory objects defined in the firmware code, which necessitates the creation of a value flow graph (VFG). The purpose of a VFG is to model the flow of data values through a program, allowing the analysis and tracking of how values are assigned, propagated, and transformed across different variables and point of the program. To construct the VFG, we use the SVF framework [38], which builds an interprocedural Memory SSA form that captures definition use chains for both top-level and address-taken variables, connecting them with value flow edges. The interprocedural static value flow analysis framework (SVF), built on LLVM, performs precise static value flow analysis of C/C++ programs. It integrates Andersen's pointer analysis to understand pointer usage and dereferencing.

Our framework extends the SVF tool [38] to develop our data graph model as follows. Once the VFG is constructed, our static framework navigates the graph, recognizes the nodes based on their roles, and labels each node with the correct ID along with dynamic information that will be sent to an external Verifier. Specifically, we identify three types of nodes in our model: *Input functions* (IN), *Use Nodes* (USE), and *Definition Nodes* (DEF). These nodes hold various pieces of information and aid in assessing the firmware's integrity status.

*Input Nodes.* To detect OOB accesses, EMBEDWATCH locates the input functions (IN) in the firmware code. These functions act as attack entry points, allowing adversaries to corrupt the firmware's internal status through OOB accesses. We need to distinguish two types of input functions based on their semantics. SIN and PIN. SIN functions return scalars (*e.g.,* `getc`), often used in loops to transfer single bytes to buffers (*e.g.,* in Listing 1). For SIN functions, we trace all written memory locations. In contrast, PIN functions directly fill buffers (*e.g.,* `scanf`) and are required to return the number of read bytes, which we use to calculate and trace the last written address. Our prototype encodes five popular IN functions (*e.g.,* `gets`, `getc`, `fgets`, `fread`, and `scanf` ), which were sufficient to handle all firmware in our experiments. However, our prototype can be extended to any function that fulfills our IN function definition, such as `uart_read` [12].

*Use Nodes.* Once we locate the IN functions, we proceed with a context-sensitive analysis based on the forward process in the interprocedural VFG [38] to detect all the possible corresponding Use nodes (USE).

There are two types of USE nodes, similar to IN functions: SUSE and PUSE. SUSE nodes store scalars in memory regions, such as "`STR R2, [R1]`" in the ARM assembly, and we trace the written memory location for these nodes. PUSE nodes manipulate buffer pointers, such as `memcpy`. We require PUSE to return the number of bytes written, just as we do with PIN, so that we can calculate and report only the last modified address. We need to distinguish SIN/SUSE from PIN/PUSE because the former are implemented as simple memory transfers (*e.g.,* `STR`), from which we can easily report each address written. For the latter, instead, we only observe the amount of memory transferred (*e.g.,* `memcpy`), thus we need to calculate the last written address. For this purpose, we design a specific instrumentation in §6.3.

*Def Nodes.* DEF nodes provide critical information about object boundaries, which can be located in various memory regions such as the stack, global sections, or heap, each with unique characteristics. For our framework, it is essential to accurately identify variable definitions along with their memory boundaries to infer out-of-bound errors (OOB) and produce meaningful crash reports. Specifically, our framework handles four types of DEF nodes: SDEF for stack variables, GDEF for global variables, HDEF for heap variables, and FDEF for fields in `structs`. For stack buffers (SDEF), which can be allocated across multiple stack frames, we further associate an index referring to the specific stack frame in which they are allocated. This information is useful for identifying the execution path that triggers the attack. For SDEF and GDEF, their sizes are extracted at compilation time. HDEF refers to `malloc` allocations, with sizes computed at runtime. FDEF allows us to infer field boundaries and detect overflows within a `struct`. This data is obtained by analyzing the function data flow: for each DEF node with a *struct type*, we reconstruct all accessed fields between the definition and the corresponding use. To achieve this, we utilize LLVM's GEP (GetElementPtr) instruction [11], which provides functionalities for accessing struct fields. By examining GEP instructions, we can precisely identify the memory structures and their fields in use, thereby determining their sizes accurately. After extracting the sizes of the definition nodes, we also need to trace their memory addresses at runtime to perform out-of-bounds (OOB) checks. Therefore, at runtime, we trace the base addresses of buffers for [G, H, S]DEF. Combining these base addresses with the buffer sizes allows us to calculate the buffer boundaries. FDEF is not traced at runtime but is included in the model M for trace verification purposes (details in §6.5).

*Linking Nodes Procedure.* After determining all node types, sizes, and base addresses, it is necessary to associate the appropriate uses (USE nodes) of variables with their definitions (DEF nodes). In the results of the application of forward-based context-aware analysis within the interprocedural VFG [38], multiple definitions of the same variable can be associated with a single use. For example, this situation arises when a function is called with parameters that are defined at different points in the code. Unfortunately, the specific definition that will be triggered at run-time cannot be statically determined due to the unpredictability of the program's execution path. To solve this problem, we initially identify all potential DEFs that the use can target as candidates. Since each entity within our analysis framework is assigned a unique ID, we can easily differentiate them. At the use point, our framework gathers not only the data pertaining to the use being executed, but also the ID of the definition targeted by the operation.

Moreover, when memory objects are passed as function arguments, our framework needs to propagate the IDs of the corresponding definitions for inter-procedural analysis. This is achieved by automatically adding an extra ID argument to each analyzed function and modifying all corresponding call sites to propagate the runtime information accordingly.

## 6.3 Firmware Instrumentation

EmbedWatch utilizes source code instrumentation to track USE and DEF operations, ensuring the Verifier can accurately detect out-of-bounds (OOB) accesses and produce reliable crash report even in the presence of potential attackers (as outlined in §4).

For every Definition (DEF) node, we monitor the buffer ID and its base address. Global Definition (GDEF) nodes are tracked during the process's initial bootstrapping phase and are logged only once. In contrast, Stack Definition (SDEF) and Heap Definition (HDEF) nodes are monitored after variable declarations by adding instrumentation points at the allocation sites and linking them to the variables that reference the allocated memory regions. This provides the Verifier with information about the buffer's location in memory. The buffer size is inferred from the source code for static allocations (*i.e.,* stack or global) or traced by instrumenting the argument of `malloc-like` functions for heap allocations. For each Use (USE) instruction, we record the instruction ID and the written address. In case of PIN/PUSE nodes, we calculate the last written address by adding the initial buffer pointer and the number of bytes written. When memory objects are passed as function arguments, EmbedWatch passes along the buffer IDs and their stack index as additional arguments. This is achieved by automatically adding extra buffer ID/stack index arguments to each analyzed function and modifying all corresponding call sites to propagate the runtime information. Moreover, we track the stack frames to differentiate between the local buffers of the functions. Specifically, the embedded device keeps a stack index that we increase and decrease at each function's prologue and epilogue, respectively. We then add the stack index as an additional field along with the other model nodes. Having the stack index helps the Verifier determine which stack frame the firmware is currently executing. To avoid tampering, we placed the stack index in a shadow area, as described further in §7.3.

## 6.4 Loop Optimization

During our experiments, we observed that certain loops caused significant runtime overhead, especially in scenarios involving large communications with the TrustZone. This was primarily due to the high number of memory operations performed by the firmware. To solve this problem, we store the USE/DEF nodes occurring within specific loops in a buffer in the Normal World. Once the loop (or the firmware) ends or when the buffer is full, we transfer the stored nodes in bulk to the Secure World. However, this optimization is only applied to loops that do not involve function calls, indirect jumps, or return instructions, as these could enable attackers to evade our system by skipping the synchronization operations with the TrustZone. In contrast, for loops without control-flow transfers, adversaries can only activate the attack payload once the loop itself terminates. This optimization approach is effective and secure for two reasons. First, tampering with the cache can only occur when the payload is activated, which happens after the loop ends and the cache has already been flushed into the TrustZone. Second, the cache is stored in a randomly allocated shadow area, similar to SafeStack [10], making it difficult for adversaries to identify its location. In our experiments, the cache mechanism significantly improved performance, yielding approximately 13.4 times better results compared to standard instrumentation (§7.2). More details on the security analysis of this solution are presented in (§7.3).

## 6.5 Trace Validation

The Verifier (represented by ❽ in Figure 1) validates the USE/DEF nodes reported by the firmware. Its purpose is to infer the firmware's memory status, detect OOBs, and produce crash reports in case of errors. The Verifier is composed of three main components: (1) one for tracking the firmware run-time status, (2) one for identifying OOBs, and (3) one for producing a crash report. We provide more details in the following sections.

*Firmware Execution Memory.* The Verifier maintains a table of active variables in the firmware based on the information received by the execution trace. In particular, for each variable, it records the following information: the stack index, node type, variable name, size, and base address. The stack index refers to the actual stack frame in which a node is executed. To minimize traced nodes and keep the table information correct, the Verifier infers the allocation of new stack frames from incoming USE/DEF traces without tracing context switch events (such as ret/call from functions). To correctly identify the new stack frame, the Verifier checks for the presence of a new DEF of a local stack variable in the trace and, if found, assumes that the previous frame is no longer in scope and removes its information from the table, reducing the amount of stored data. This policy is implemented in the algorithm line 7. After obtaining the variable name, node type, base address, size, and stack depth from the trace, the system updates the table (see line 15). This

**Algorithm 6.1:** Variable Alive Update

```
   Input: Model M
   Input: Runtime Node d
 1 stack_index ← d.stack_index;
 2 node_type ← d.node_type;
 3 if d is SDEF ∨ d is GDEF then
 4     size ← get_size(M, d.id);
 5     var_name ← get_var_name(M, d.id);
 6     base_address ← d.base_address;
 7 else if d is HDEF then
 8     size ← d.size;
 9     var_name ← get_var_name(M, d.id);
10     base_address ← d.base_address;
11 end if
12 if table_has(stack_index, var_name) then
13     delete_variables(stack_index, var_name);
14 end if
15 insert_table(stack_index, node_type, var_name, base_address, size);
```

**Algorithm 6.2:** OOB Detection

```
   Input: Model M
   Input: Runtime Node u
 1 stack_index ← u.stack_index;
 2 address ← u.address;
 3 if u has [DEF_ID, DEF_index] then
 4     d ← get_defintion(M, u.DEF_ID);
 5     var_name ← d.var_name;
 6     var_stack_index ← u.DEF_index;
 7 else
 8     d ← get_defintion(M, u.id);
 9     var_name ← d.var_name;
10     var_stack_index ← stack_index;
11 end if
12 var_base_address ← get_var_base_address(M, var_name, var_stack_index);
13 var_size ← get_var_size(M, var_name);
14 if oob_detected(store_address, var_base_address, var_size) then
15     vulnerability_analysis(var_stack_index, stack_index, u, d);
16 end if
```

approach ensures that the information on the firmware variables is always accurate and up-to-date.

*OOB Detection.* The Verifier can infer the boundaries of objects involved in USE and IN operations and detect Out-Of-Bounds (OOB) by using the alive variables table. The algorithm is described in Algorithm 6.2. For each run-time node `u`, the Verifier obtains its stack index and the written address (line 4-6). If `u` contains a DEF_ID and DEF_index, it means that the node is operating on a pointer passed as a reference. In that case, the Verifier extracts the buffer definition, its variable name, and stack index from the node `u` itself (line 4 and 6). If not, the buffer definition and the variable name are obtained from the Use-Def graph (line 8), while the stack index remains the same as the incoming node `u` (line 10). Once the variable name and its stack frame have been inferred, the Verifier checks if the address falls within the buffer boundaries by enquiring the table of alive variables . The function `oob_detected` also detects OOB in `struct` fields. If `d` refers to a `struct`, the Verifier searches for FDEFs between the DEF associated with `var_name`/`var_stack_index` and `u` in the Use-Def graph. For each traversed FDEF, the Verifier can infer the field or subfield that the firmware is pointing to, and compute its boundary. Finally, if an OOB is detected, the Verifier performs a vulnerability analysis (line 15).

*Crash Report.* Once the Verifier detects an OOB, the function `vulnerability_analysis()` produces a report on the anomaly (line 15). In particular, the Verifier reports: (1) the line of code where the OOB occurred, (2) the object involved in the OOB (*i.e.,* buffer or `structs` field) and (3) the stack trace. The stack trace is reconstructed from the table of living variables. In particular, knowing the stack frame where a variable has been allocated (*i.e.,* `var_stack_index`), and the stack frame of the USE node (`stack_index`), allow us to infer the last functions alive in the firmware by inspecting the variables alive table. This reveals contextual information to assess and patch the observed bug.

## 7 EVALUATION

We assess the properties of EmbedWatch over *three* main research questions: (**RQ1**) Does EmbedWatch produce correct vulnerability analysis reports (Efficacy Analysis) (§7.1)? (**RQ2**) Can EmbedWatch be deployed in real platforms (Performance Evaluation) (§ 7.2)? (**RQ3**) What are the security property of EmbedWatch (§7.3)?

*Prototype Details.* All experiments are carried out on an ARM Pi 3b + commercial board, which is equipped with ARM TrustZone support and compatible with the OP-TEE commit `10b7b60b`. The selected board mounts a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit processor @ 1.4GHz, and 1GB DDR2 RAM. We selected this board as our reference device for prototyping because it is a widely used IoT development board. The Trusted OS (OPTEE-OS) runs in Secure-Mode (SEL-1), the rich OS (BusyBox) is shipped with OP-TEE. For the compilation module and model generation, we extend the LLVM version 13 and use the SVF commit `cc3151c0`.

*Firmware Dataset.* We assess the EmbedWatch system by evaluating a total of 15 open source embedded programs [3]. Among these, two firmwares were sourced from the OAT dataset [39], which comprises a total of 5 firmwares that were used to test the OAT framework. We omit three firmware from the OAT dataset due to the manual effort required to port it to our development board, a result of the device-specific nature of embedded programs rather than any limitation of our system. We selected nine firmwares from well-known open source repositories—Arduino Projects [1] and Raspberry Pi Pico Examples [9]—and the remaining four from the Fuzzware Open Source Data Set Fuzzware [34]. As for Fuzzware, we select firmware that already exposes spatial memory errors, making them ideal for testing EmbedWatch's capabilities as they contain *seven* real vulnerabilities, showcasing its efficacy in practical scenarios. The chosen firmware projects offer diversity, reflecting the simplicity often encountered in various embedded programs. When needed, we make modifications to the firmware source code to ensure library compatibility and support different architectures. More in detail for our experiments, we created two datasets: $D_p$ for evaluating performance and false positive rate, and $D_s$ for testing the system's effectiveness. For $D_p$ we carefully selected 11 embedded programs that exhibit the same behavior of polling sensors and performing specific actions on the data retrieved. This standardized approach allows us to create a reliable and consistent benchmark for our experiments. $D_s$, instead, consists of 15 programs, including the 11 programs from $D_p$, deliberately modified to introduce specific vulnerabilities, with one vulnerability introduced for each firmware. Additionally, the four remaining Fuzzware firmwares already contain genuine spatial memory vulnerabilities. More detailed information on the types of vulnerability introduced in the firmware can be found in §7.1.

## 7.1 RQ1: Bug Analysis & False Positives

*Synthetic Vulnerabilities.* To rigorously assess EmbedWatch's capabilities in detecting attacks and performing vulnerability analysis, we deliberately inject *eleven* various types of vulnerabilities into the system's source code (*11* firmware), which include: (1) Control-Flow Hijacking: We overwrite stack variables to disrupt the firmware's control-flow. (2) Global Variable Corruption: Intentional corruption of global variables within the code. (3) Struct Field Overwrite: The specific fields in `struct` data structures are overwritten. (4) Heap Variable Overwrite: Targeting and modifying variables residing in the heap. Our objective is twofold: to evaluate EmbedWatch's resilience against both control-flow attacks, which manipulate process execution flow, and data-only attacks that exploit a program without altering its execution flow. To thoroughly examine vulnerability analysis, we strategically position vulnerabilities and attack points randomly in different code locations. The vulnerability data set is documented in [3]. To reach and trigger these vulnerabilities, we utilize a symbolic engine called angr [37], which generates the attack inputs for each firmware.

For each attack, EmbedWatch successfully detects the overflow attack and provides the exact point in the source code where the vulnerabilities were injected, along with the buffer involved in the operation. We manually inspect the execution trace to validate the crash report. Our experiment demonstrates the fundamental functioning of our prototype and confirms that EmbedWatch is a viable method for remote verifiers, offering precise vulnerability analysis. In addition, we conducted a thorough evaluation of false positives in our system by running our firmware data set with the input sets outlined in Table 1. The test inputs were generated using a symbolic executor using a code coverage metric. We closely monitor the generated execution traces of more than 1,300 inputs and observe no instances of false alarms.

*Real-World Vulnerabilities.* Utilizing the Fuzzware dataset's crash analysis section [6], we carefully selected four firmware samples: Heat Press, PLC, thermostat, and rf door lock. These chosen firmwares contain *seven* spatial memory errors, with heat press and plc exhibiting overflows on a field attribute, while the others suffer from stack buffer overflows. Although the Fuzzware data set provides input to trigger vulnerabilities, we enhanced the source code with instrumentation, leading to slightly different memory alignments compared to the original firmware. To address this, we used the QEMU system for debugging and identifying an optimal new input set that worked effectively with our instrumentation.

Deploying the firmware on our board, we validated EmbedWatch's capability to accurately infer the root bug code location. EmbedWatch detected all spatial memory errors and automatically generated identical crash reports as those already reported in the Fuzzware dataset. This experiment significantly showcases EmbedWatch's effectiveness in detecting real-world vulnerabilities across different classes, including dynamic memory allocation, global buffer, and local buffer vulnerabilities.

## 7.2 RQ2: System Usability & Performance

*Unit Test.* To thoroughly assess the reliability and comprehensiveness of EmbedWatch, we conduct an extensive evaluation by generating a large corpus of inputs and their corresponding expected outputs. To ensure an unbiased input generation process, we employ symbolic execution, which allows us to automatically create safe inputs that uniformly cover various code sections within the firmware. Our objective is to maximize the coverage of the code in terms of the executed instructions for $D_p$. For this purpose, we rely on angr [37], which provides automatic code coverage input generation. Table 1 details statistics regarding the inputs generated.

*Macrobenchmark.* As a crucial performance metric, we assess the influence of our solution on the run-time performance of the firmware. Specifically, we conducted three sets of measurements: one without EmbedWatch (baseline), another with EmbedWatch, and a third with EmbedWatch along with our caching mechanism. These measurements enable us to present the columns "Performance Overhead", vividly demonstrating that our approach introduces an average overhead of approximately 1.44%, which diminishes to around 0.23% when the cache is applied. We set the cache size to 128 elements as a representative value, while conducting an ad hoc experiment on the cache performance in the following paragraphs. The results show that EmbedWatch incurs an overhead that range (0.07% - 36.04%), geometric mean 1.4 +0.4% encryption protocol without cache, whereas enabling the cache drops the overhead to range (0.01% - 2.33%), geometric mean 0.228 +0.4% encryption protocol . Further investigation of the sources of overhead reveals two different scenarios. In some firmwares, such as clock_phone, ledmatrixpainter, and hue-motion, the overhead is proportional to the number of instrumentation points executed at runtime. This relationship between overhead and the number of instrumented variables can be estimated by the results of our framework analysis (§6.2). The more instrumentation points are present, the greater the overhead. However, there are firmwares that do not follow this trend (*e.g.,* syringe_pump), which exhibits limited overhead despite a high number of instrumentation points. This is because these firmwares make use of idle operations, such as sleep or loops, that extend and skew the execution time. For example, syringe_pump uses a sleep function for liquid calibration. The use of idle operations in the firmware is a common practice, as many sensor data are read by waiting for events or physical components to complete. The usage of idle operations accounts for the discrepancy between the number of executed instrumented points and the relative overhead in some firmware. Finally, Table 1 shows the average trace validation time for each firmware based on the selected inputs, the trace validation is calculated with firmware without cache to show the worst-case scenario. We also implemented communication encryption to guarantee the confidentiality and integrity of the data exchanged between the firmware and the remote verifier. This approach protects against potential in-transit data manipulation and can detect unauthorized access by malicious attackers. The EmbedWatch protocol uses the AES CBC symmetric encryption algorithm with a 128-bit key size. The key is preloaded onto the device during the deployment phase and is also stored in the Verifier. As illustrated in Table 1, encryption introduces a small overhead of 0.4% in firmware execution time compared to the unencrypted counterpart. It is noticeable that firmware with larger TrustZone interaction and lower execution time has higher overhead in the encryption stage because it has more data transit to the verifier that needs to be encrypted.

*Microbenchmark.* The goal of this evaluation is to identify the main source of overhead in EmbedWatch. This overhead is caused by the interaction between the firmware and the Trusted Application (TA) during run-time tracing. The firmware uses a trampoline library to call two secure functions, `lib_def_use` and `lib_flush_cache`. The first function, `lib_def_use`, tracks normal USE/DEF/IN interactions and transfers a small amount of data to the TA. The second function, `lib_flush_cache`, is used for cache optimization and performs bulk loads into the TA. We execute each secure function of the TA 1,024 times and compute the average overhead. Since the execution of `lib_flush_cache` varies according to the cache size, we repeat the experiments with 32, 64, 128, 256, 512, 1,024, 2,048, and 4,096 entries in the cache.

The overall execution time of the function `lib_def_use` was found to be 1,612 ms with a standard deviation of 0.60. In contrast, the overhead of the function `lib_flush_cache` depends on the cache size, which ranges from 1,548 to 3,298 ms. In the worst-case scenario, having 4,096 entries takes approximately twice as long as transferring a single entry with `lib_def_use`. The difference in performance is attributed to the context switch, which is the main source of overhead for EmbedWatch.

*Firmware Size.* The firmware size increase results from the addition of instrumentation points and trampoline library calls during compilation. On average, a 40% increase in size is observed, contrasting with the lower overhead exhibited by OAT (17%) in terms of instrumentation points. This discrepancy is attributed to the additional code implemented to minimize runtime overhead through instrumentation. We were unable to replicate the BLAST results due to the lack of source code. However, BLAST [45] reported a 64% increase in size (best case w/o function inlining) with embench-iot dataset [5]. In general, we consider 40% increase reasonable, thus positioning our approach midway between OAT, which enforces partial CF attestation, and BLAST, which achieves higher precision through whole-program CF attestation.

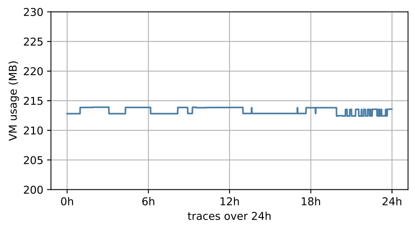


**Figure 3: Virtual Memory (VM) allocated by the Verifier after 24h of interaction with a firmware.**

*Verifier Performance.* To assess the long-lasting stability of EmbedWatch, we carried out a comprehensive 24 hour experiment, running the Verifier continuously while a firmware continuously generated and sent traces. For this particular evaluation, we deliberately chose xml-parser as our test subject, given its participation in extensive memory operations, including stack and heap allocations, making it an optimal candidate for rigorous testing.

Throughout the experiment 3, the Verifier efficiently processed incoming traces without any noticeable delays. Regarding memory allocation, we observed that the Verifier quickly stabilized at a usage of 215 MB. Verifier memory consumption depends on factors such as the depth of the firmware call stack and the number of new memory chunks allocated, as discussed in §6. This experiment demonstrates the Verifier's stability during extended runs. Considering our intention to deploy the Verifier on high-performance machines, the allocation of 215 MB of memory is considered reasonable.

*Performance Comparison with State-of-the-Art.* As EmbedWatch is designed for spatial memory errors detection leveraging remote attestation of ARM-based embedded devices, our primary points of comparison are OAT [39] and BLAST [45]. Although there are significant differences in the properties being attested — with EmbedWatch focusing on spatial memory violation detection and OAT and BLAST aiming to ensure control flow integrity — this performance comparison is still extremely valuable. We demonstrate that the overhead associated with our system is in line with that of other embedded system detectors.

Given that BLAST lacks open source code, our comparison was confined to OAT. We based our evaluation on the number of TrustZone invocations, both statically inserted during the compilation phase and dynamically triggered during the testing phase. These data serve as a suitable basis for comparison, since both frameworks exhibit a similar overhead in their interactions with the secure world. To collect these data, which are reported in Table 2, we modify the OAT trampoline libraries to count interactions with the Trusted Application (TA). This allows us to streamline the emulation process while retaining the data of interest. In addition, we add simple trace capabilities. Subsequently, we build the OAT's compiler, the trampoline libraries, and test our dataset compiled with OAT tools. We exercise the firmware code with the same inputs as listed in Table 1 and record the number of instrumented points hit by the execution.

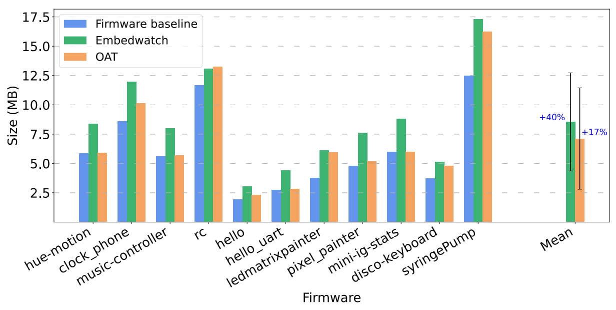


**Figure 4: Comparison of instrumented binary size between EmbedWatch and OAT.**

The results show that EmbedWatch exhibits a significantly lower number of calls compared to OAT. The disparity arises because the OAT is heavily dependent on TrustZone calls to collect information

**Table 1: Number of dynamic instrumentation points and performance overhead. We measure the firmware execution time with cache enabled and with cache+encryption.**

| Firmware | Input | Dyn. Instr. | | | Performance overhead | | | | Trace | Verify | | Execution time | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | w/o cache (%) | | w/ cache (%) | | (byte) | (sec) | | (msec) | w/encryption | encryption |
| | | DEF | USE | Total | mean | std. dev. | mean | std. dev. | mean | total | single entry | w/cache | w/cache (msec) | overhead (%) |
| pixel_painter | 43 | 6 | 13 | 19 | 0.64 | 0.0097 | 0.16 | 0.0079 | 254 | 0.05 | $8 \times 10^{-5}$ | 866.52 | 868.03 | 0.1742 |
| clock_phone | 10 | 1363 | 470 | 1833 | 36.04 | 42.2026 | 2.33 | 2.6471 | 292 | 0.05 | $5 \times 10^{-5}$ | 1300.4 | 1343.06 | 3.2810 |
| ledmatrixpainter | 100 | 46 | 100 | 146 | 10.06 | 0.0756 | 0.74 | 12.3331 | 2314 | 6.10 | $7 \times 10^{-3}$ | 603.77 | 604.41 | 0.1060 |
| rc | 10 | 50 | 51 | 101 | 0.23 | 0.1258 | 0.01 | 0.0025 | 5780 | 0.52 | $2 \times 10^{-5}$ | 10893.07 | 10895.39 | 0.0212 |
| hello | 3 | 26 | 2 | 28 | 1.26 | 0.3664 | 0.32 | 0.0057 | 448 | 0.13 | $7 \times 10^{-5}$ | 10893.07 | 10895.39 | 0.0212 |
| mini-ig-stats | 3 | 12 | 33 | 45 | 1.79 | 0.2223 | 0.22 | 0.0250 | 224 | 0.05 | $6 \times 10^{-5}$ | 722.77 | 723.79 | 0.1411 |
| disco-keyboard | 49 | 4 | 11 | 15 | 0.07 | 0.0058 | 0.00 | 0.0054 | 276 | 0.06 | $8 \times 10^{-5}$ | 1017.95 | 1019.43 | 0.1453 |
| hello_uart | 5 | 3 | 8 | 11 | 0.35 | 0.0088 | 0.12 | 0.0156 | 206 | 0.22 | $34 \times 10^{-5}$ | 1053.50 | 1056.00 | 0.2373 |
| music-controller | 1024 | 240 | 632 | 872 | 2.04 | 0.0031 | 0.05 | 0.0007 | 35313 | 3.60 | $3 \times 10^{-5}$ | 10104.49 | 10107.12 | 0.0260 |
| syringe_pump | 10 | 1474 | 895 | 2369 | 0.38 | 0.2113 | 0.00 | 0.0000 | 2427 | 0.30 | $4 \times 10^{-5}$ | 99.10 | 149.40 | 50.7568 |
| hue-motion | 46 | 64 | 88 | 152 | 19.31 | 8.0887 | 1.44 | 0.4508 | 143 | 0.05 | $9 \times 10^{-5}$ | 402.93 | 404.52 | 0.3938 |
| **Geometric mean** | - | 45.163 | 52.863 | 111.625 | 1.411 | 0.122 | 0.228 | 0.035 | 784.603 | 0.211 | $9.6 \times 10^{-5}$ | 711.46 | 751.46 | 0.4076 |

**Table 2: Comparison of both static and dynamic calls to trampoline libs of OAT and EmbedWatch. For "Dynamic (TZ calls)", we compute the arithmetic average of the trampoline invocations during the testing phase.**

| Firmware | OAT | | EmbedWatch | | |
|---|---|---|---|---|---|
| | Static | Dynamic (TZ calls) | Static | Dynamic (trampoline calls) | Dynamic (TZ calls) |
| heat_press | 52 | 234.7 | 10 | 33 | 6.0 |
| music-controller | 36 | 1176.6 | 30 | 872 | 15.0 |
| pixel_painter | 22 | 160.0 | 33 | 19 | 6.0 |
| rc | 57 | 1743.0 | 10 | 101 | 6.0 |
| syringePump | 75 | 1730709.6 | 23 | 2370 | 106.8 |
| clock_phone | 37 | 120.0 | 48 | 1833 | 94.2 |

on conditional branches within the firmware code, which is essential for accurate CFI attestation, as reported in [39]. In contrast, our system focuses solely on tracking the memory structures that depend on the input.

In our case, it is crucial to acknowledge that every time a call to the TrustZone is made, we pass more data to the TA compared to the OAT system. This arises from the presence of the cache, which effectively reduces the number of context switches with the TA, representing the primary source of overhead in such systems. The benefit of using the cache has already been studied in §7.2. We report the instrumentation without cache (trampoline calls) and with cache (TZ calls) in Table 2. Similarly to §7.2, we set the cache to 128 elements. We conclude that EmbedWatch delivers efficiency on par with the leading state-of-the-art solutions in the embedded systems landscape.

## 7.3 RQ3: Security Analysis

Our threat model takes into account the possibility that attackers discover and exploit previously unknown vulnerabilities in the embedded programs running in the Normal World. However, we assume that code injection or modification is not possible, which is already prevented by existing code integrity schemes for embedded devices [18, 27]. To bypass EmbedWatch, an attacker would have to take one of the following actions: (a) disable the instrumentation or trampolines, (b) misuse the instrumentation parameters, (c) exploit the interfaces that the measurement engine exposed to the trampolines, and (d) alter the execution trace, including replicating a previous attack. In the following, we examine each case separately.

(a) To disable the instrumentation, the attacker would need to modify the code, which is not possible because we assume that code integrity measures are in place. To evade the instrumentation, the attacker would need to take control of the control flow of the application, but since the instrumented code is executed before such actions, it is recorded in the execution trace.

(b) To successfully compromise the parameters passed to the trampoline functions, an attacker would need to overwrite the node IDs, the stack index, the buffer base address or the buffer size (for HDEF). Our instrumentation is designed to protect against attacks against these components (§6.3). We store the values of statically inferred information, such as node IDs, as immediates in the read-only section `.text` of the program during compilation. The values of runtime attributes, such as the base address and size of a buffer, are recorded in the corresponding DEF events immediately after the buffer is created. Therefore, we store the base address and size in the TA before an overflow or attack occurs. For IDs that refer to buffers passed as function pointers and the stack index, we adopt a solution similar to SafeStack [10]: a randomly allocated area that contains sensitive variables (*i.e.,* buffer ID and stack index). We also employ a similar solution to SafeStack for protecting loop optimization. In this case, we maintain a cache that is randomly allocated in the Normal World, which is flushed in the TA immediately after the loop ends. Since we assume that the loop does not contain indirect jumps or function calls, a payload cannot attack the buffer before the loop terminates, while an overflow would not easily reach the cache.

(c) Implementing our compiler comes with a unique restriction: it disallows the use of SMC (Secure Monitor Call) instructions outside the trampoline library. This restriction is put in place to ensure the security and integrity of the system. It allows only the trampoline functions to invoke the Trusted Application (TA) interfaces, effectively disabling the Normal World from accessing them. By implementing this restriction, we can maintain the secure environment for the system and prevent unauthorized access to the TA interfaces.

(d) Our system ensures the integrity of the attestation blob through a signature generated by TEE using a hardware-provisioned private key. A verifier can easily check the signature and verify the integrity of the attestation blob. Replay attacks are also prevented by checking if the cryptographic nonce inside the attestation blob matches the one originally generated by the verifier.

## 8 DISCUSSION

In this section, we discuss limitations of EmbedWatch and propose new challenges for future work.

*Aliasing of local variables.* The EmbedWatch prototype tracks buffer IDs for variables passed as function arguments, including their aliases during runtime. However, it currently does not handle scenarios where a static pointer is linked to multiple local variables. To address this, we need to extend buffer ID propagation to local variables, detecting aliases at compile time, similar to function pointer parameters. In our experiments, we have not encountered any such cases, so the implementation of this feature is deferred to future improvements. However, it remains an important area of focus for further development.

*Interrupts handling.* The current prototype does not integrate the interrupt handler trace. We can easily extend EmbedWatch to handle signals by using our graph model to analyze signal handlers that contain attacker-controllable memory structures. Then the existing EmbedWatch design already identifies and reports OOB accesses within the signal handlers. Therefore, we consider the extension for interrupt handlers as a pure engineering task. Moreover, the current target firmware does not contain interrupt handlers with attacker-controllable input.

*Temporal memory safety.* EmbedWatch is primarily designed to identify and address exploitable spatial memory safety violations. However, we propose that its methodology and framework could be adapted to also recognize exploitable temporal memory safety violations. This adaptation, while promising, introduces considerable challenges. A key obstacle is the need to monitor the liveness of dynamic memory allocations, a vital component in detecting temporal errors. Such monitoring, as previously discussed, has the potential to adversely affect system performance. Additionally, the development of an advanced pruning system is essential. This system should mirror the functionality of the one in EmbedWatch, selectively instrumenting operations that pose a risk of leading to a security breach. This targeted approach is crucial to ensure the feasibility of the enhanced system in practical, real-world applications. The full implementation of this extension requires further research and development, representing a significant yet worthwhile direction for future advancements.

## 9 CONCLUSION

In this paper, we introduce EmbedWatch, a novel crash reporting system designed for embedded devices. By combining fat pointer mechanisms with remote attestation, it addresses the distinct challenges of attack analysis in resource-constrained and real-time environments. Our system excels in detecting and analyzing memory errors, such as data-only and control-flow attacks, across different memory segments. Our experimental evaluations of real-world firmware and CWEs show that EmbedWatch is very accurate in identifying the root causes of spatial memory errors, with a minimal overhead range of range (0.01% - 2.33%), geometric mean 0.228 +0.4% encryption protocol . These findings underscore the system's effectiveness and suitability for real-world IoT environments.

## ACKNOWLEDGMENTS

This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.
This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation.

## REFERENCES

[1] [n. d.]. Arduino project. https://github.com/mattiasjahnke/arduino-projects.
[2] [n. d.]. CrashReporting. https://wiki.ubuntu.com/CrashReporting.
[3] [n. d.]. EmbedWatch dataset. https://github.com/laserunimi/embedwatch/tree/master/test.
[4] [n. d.]. EmbedWatch source code. https://github.com/laserunimi/embedwatch.
[5] [n. d.]. Embench: Open Benchmarks for Embedded Platforms. https://github.com/embench/embench-iot.
[6] [n. d.]. Fuzzware Dataset. https://github.com/fuzzware-fuzzer/fuzzware-experiments.
[7] [n. d.]. Intel PT. https://www.intel.com/content/www/us/en/developer/videos/collecting-processor-trace-in-intel-system-debugger.html?wapkw=intel.
[8] [n. d.]. Mozilla Crash Reporter. https://support.mozilla.org/en-US/kb/mozillacrashreporter.
[9] [n. d.]. Raspberry Pi pico-example. https://github.com/raspberrypi/pico-examples.
[10] [n. d.]. SafeStack. https://clang.llvm.org/docs/SafeStack.html.
[11] [n. d.]. The Often Misunderstood GEP Instruction. https://llvm.org/docs/GetElementPtr.html.
[12] [n. d.]. TI-RTOS Drivers. https://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/tirtos/2_20_01_08/exports/tirtos_full_2_20_01_08/products/tidrivers_cc13xx_cc26xx_2_20_01_10/docs/doxygen/html/_u_a_r_t_8h.html.
[13] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 743–754.
[14] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. 2019. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems.. In *NDSS*.
[15] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*. https://api.semanticscholar.org/CorpusID:14320211
[16] Todd M Austin, Scott E Breach, and Gurindar S Sohi. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*. 290–301.
[17] Steve Christey, J Kenderdine, J Mazella, and B Miles. 2013. Common weakness enumeration. *Mitre Corporation* (2013).
[18] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting Bare-Metal Embedded Systems with Privilege Overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*. 289–303. https://doi.org/10.1109/SP.2017.37
[19] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.
[20] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 17–32.
[21] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. 2018. Litehax: lightweight hardware-assisted attestation of program execution. In *2018*

*IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[22] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. 2017. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[23] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers.. In *NDSS*, Vol. 17. 1–15.

[24] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th USENIX Security Symposium (USENIX Security 19)*. 1787–1804.

[25] ARM Holdings. 2009. ARM security technology: Building a secure system using trustzone technology. *Retrieved on June* 10 (2009), 2021.

[26] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: a safe dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[27] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *NDSS*.

[28] Tim Kornau. [n. d.]. Return oriented programming for the ARM architecture.. In *Ph.D thesis*.

[29] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. 2019. POMP++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1929–1942.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.* 44, 6 (jun 2009), 245–258. https://doi.org/10.1145/1543135.1542504

[31] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[32] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on {ARM}. In *26th USENIX Security Symposium (USENIX Security 17)*. 33–49.

[33] Takamichi Saito, Ryohei Watanabe, Shuta Kondo, Shota Sugawara, and Masahiro Yokoyama. 2016. A survey of prevention/mitigation against memory corruption attacks. In *2016 19th International Conference on Network-Based Information Systems (NBiS)*. IEEE, 500–505.

[34] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1239–1256. https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[36] Jiameng Shi, Wenqiang Li, Wenwen Wang, and Le Guan. [n. d.]. Facilitating Non-Intrusive In-Vivo Firmware Testing with Stateless Instrumentation. ([n. d.]).

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok: (State of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.

[38] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[39] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1433–1449.

[40] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[41] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. 2019. {ScaRR}: Scalable Runtime Remote Attestation for Complex Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 121–134.

[42] Flavio Toffalini, Mathias Payer, Jianying Zhou, and Lorenzo Cavallaro. 2022. Designing a Provenance Analysis for SGX Enclaves. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 102–116.

[43] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 529–540.

[44] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*. 17–32.

[45] Nikita Yadav and Vinod Ganapathy. 2023. Whole-Program Control-Flow Path Attestation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2680–2694. https://doi.org/10.1145/3576915.3616687

[46] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *5th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*.

[47] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. 2017. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 384–391.

[48] Yumei Zhang, Xinzhi Liu, Cong Sun, Dongrui Zeng, Gang Tan, Xiao Kan, and Siqi Ma. 2021. ReCFA: resilient control-flow attestation. In *Annual Computer Security Applications Conference*. 311–322.

[49] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4345–4363. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen