Computers & Security 170 (2026) 105029

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full Length Article

REMOTA: Remote attestation for detecting use-after-free in low-power microcontrollers

Matteo Zoia [a,*], Mirco Picca [a], Davide Rusconi [a], Andrea Monzani [a], Flavio Toffalini [b], Danilo Bruschi [a], Andrea Lanzi [a]

[a] University of Milan, Milan, Italy
[b] Ruhr-Universität Bochum, Germany

ARTICLE INFO

Keywords:
Embedded system
Memory errors
Use-after-free
Remote attestation

ABSTRACT

In this paper, we introduce REMOTA, a novel remote attestation protocol to capture dynamic memory allocations and uses in microcontroller embedded systems, enabling detection of use-after-free errors. REMOTA performs a precomputation analysis to identify a minimal set of key points in the control flow graph, called checkpoints, which serve as boundaries enclosing sequences of pointer operations that occur along the same execution path. These checkpoints allow the grouping of multiple pointer usages into larger, semantically meaningful units, enabling efficient and targeted instrumentation. This approach is particularly effective in resource-constrained environments, as it minimizes runtime overhead while offloading verification to a remote server. REMOTA incorporates a remote verifier that receives information from the executing firmware and replicates instructions to dynamically reconstruct pointer usage and emulate memory state, allowing lightweight use-after-free detection. Through the evaluation of real-world firmware on an STM32 microcontroller, REMOTA demonstrates high precision 100% with low overhead, geometric mean 4.47% on the tested dataset. Its scalability and efficiency make REMOTA a practical solution for securing resource-constrained embedded devices in production environments.

1. Introduction

The widespread deployment of the Internet of Things and embedded devices in critical infrastructure, healthcare, and consumer markets requires robust security mechanisms. However, these devices lack the computation power and memory needed to run established IT cyber threat protections. A malicious attacker can exploit vulnerabilities to compromise device functionality, steal sensitive data, or even cause physical harm. The main significant challenge in securing these devices lies in designing solutions that ensure security properties while preserving devices' performances. This objective underscores the critical need for effective security mechanisms that can continuously monitor such devices to detect malicious activities. As embedded systems are low-level devices where C and C++ are the primary unsafe languages used for firmware development, memory-related errors, including use-after-free, buffer overflows, and memory leaks, represent a significant category of vulnerabilities.

Earlier attempts to develop detection systems to address memory errors in embedded systems have often relied on remote attestation strategies, integrating methods such as Control-Flow Integrity (CFI) (Abera et al., 2016; Sun et al., 2020; Yadav and Ganapathy, 2023a) or fat-pointer system (Rusconi et al., 2024). However, current remote attestation protocols do not model dynamically allocated heap object. Consequently, adversaries can still rely on primitives based on *temporal bugs*, such as Use-After-Free (UAF), to take over embedded devices.

Recent real-world attacks have demonstrated that UAFs vulnerabilities are actively exploited in embedded systems. For example, CVE-2024-23923 enabled zero-click remote code execution on the Alpine Halo9 infotainment system via a UAF in its Bluetooth stack, while CVE-2024-40885 affected Intel's M20NTP BIOS firmware, and CVE-2022-48695 exposed a UAF in the Linux kernel firmware. These cases demonstrate the practical impact of UAFs in embedded environments and motivate the need for lightweight runtime verification techniques for resource-constrained systems.

UAF vulnerabilities have been extensively studied in desktop, kernel, and server contexts; the fundamental nature of these bugs remains the same on microcontrollers: dereferencing a pointer to a previously freed object (e.g., aliasing and dangling pointer issues). What changes significantly, however, is the feasibility of deploying defenses

* Corresponding author.
E-mail address: matteo.zoia@unimi.it (M. Zoia).

https://doi.org/10.1016/j.cose.2026.105029
Received 8 October 2025; Received in revised form 4 May 2026; Accepted 15 June 2026
Available online 6 July 2026

in resource-constrained environments. Most of the existing solutions, such as FreeWill (He et al., 2022), FFmalloc (Wickman et al., 2021), RTT-UAF (Du et al., 2024), and DangZero (Gorter et al., 2022), are designed under assumptions that are fundamentally misaligned with the characteristics of microcontroller environments. These tools rely on features like shadow memory, heavy dynamic binary instrumentation, or virtual memory page protections, all of which require substantial computational resources and hardware support. For instance, DangZero leverages direct page table access, and FFmalloc prevents memory reuse through a custom allocator—all of which are infeasible on microcontrollers due to their tight constraints on memory, CPU, and energy consumption or compatibility. In general these approaches are highly effective on general-purpose systems, but their assumptions, such as the availability of virtual memory, page tables, or large RAM budgets, do not hold on bare-metal microcontrollers.

Microcontrollers typically lack an MMU, use a flat memory space, and run bare-metal firmware or lightweight RTOSs without process isolation. With only tens to hundreds of kilobytes of RAM, they cannot afford large metadata or heavy runtime sanitization. Consequently, reusing desktop-class sanitization frameworks would result in excessive performance and memory overhead, making them unsuitable for production on such platforms.

Recently, security experts have moved toward software testing to detect use-after-free (UAF) bugs during development. In this regard, IPEA (Shi et al., 2024) represents an advanced sanitizer for microcontrollers. This work decouples the collection of runtime execution data from remote bug detection. Moreover, it leverages software memory tagging, associating metadata (tags) with memory operations to detect unauthorized or incorrect memory use. This enables IPEA to identify common vulnerabilities such as use-after-free, buffer overflows, and memory corruption. While IPEA effectively exposes memory errors during testing, it has two critical limitations that prevent its use as a standalone remote attestation protocol. First, IPEA introduces substantial performance overhead due to extensive instrumentation making it impractical for deployment in production environments. Second, and most important, IPEA lacks a security layer to defend against active attacks, limiting its use to pre-deployment analysis rather than real-time protection. In response to these challenges, we introduce RemOTA, an innovative framework designed to detect Use-After-Free temporal memory error in embedded devices through remote attestation. Our solution employs a novel algorithm to identify an optimal set of instrumentation points, known as checkpoints, to establish a remote attestation channel with a remote verifier. Specifically, instead of instrumenting every pointer operation as seen in memory tagging approaches, our algorithm identifies strategic points in the control flow graph (CFG) of the firmware. These checkpoints act as flow separators that group together multiple pointer *uses* occurring sequentially along the same execution path, up to the next control-flow divergence. The checkpoints are postdominator nodes of pointer *use*-sequences along a given path. By instrumenting only these checkpoints, the system can provide the verifier a small trace of the executed paths, reducing communication overhead. Upon receiving checkpoint data, the verifier reconstructs the execution trace and emulates the memory allocation operations executed along the path. By delegating memory operation emulation to the verifier, our strategy lowers the number of necessary instrumentation points within the firmware, thus enhancing performance and minimizing run-time overhead on the microcontroller.

In order to evaluate RemOTA, we performed a series of experiments aimed at evaluating the runtime overhead and testing the security guarantees. Our findings indicate that RemOTA demonstrates high accuracy in pinpointing bugs such as UAF in the examined firmware. This paper makes the following contributions:

- We introduce RemOTA, a remote attestation specifically designed to track heap object and detect temporal memory errors Use-After-Free in embedded microcontroller devices. Our solution addresses several technical challenges unique to resource-constrained environments and is implemented as a fully functional prototype. To foster reproducibility and further research, the complete source code of RemOTA, along with the evaluation datasets, is publicly available at Zoia et al. (2026).
- We present a novel static framework, the allocation graph model, coupled with a simulation memory model, for precise detection of Use-After-Free bugs. This framework optimizes firmware analysis by leveraging strategic node selection and runtime data and seamlessly integrates with the SVF framework (Sui and Xue, 2016), surpassing the analytical precision of existing methodologies.
- We evaluated RemOTA on the ARM-based STM32 NUCLEO-L552ZE-Q real board using real-world IoT firmware and a dataset of known temporal memory vulnerabilities. RemOTA achieved a precision of 100% in detecting temporal errors, with a low runtime overhead (geometric mean 4.47%) and minimal memory usage—demonstrating its suitability for deployment in resource-constrained embedded systems.

2. Related work

Remote Attestation and Embedded System Security. Prior efforts in remote attestation, such as C-FLAT (Abera et al., 2016), OAT (Sun et al., 2020), EmbedWatch (Rusconi et al., 2024), and kAGE (Du et al., 2022) laid the foundation for verifying control-flow integrity in embedded systems. Other systems protect the firmware code by using compartmentalization (Clements et al., 2018) or implementing shadow stack protection (Almakhdhub et al., 2020; Zhou et al., 2020). These systems primarily focus on spatial memory errors or control-flow deviations but do not address temporal memory bugs like Use-After-Free (UaF). Similarly, IPEA (Shi et al., 2024) proposed a microcontroller sanitizer for test environments by decoupling live information extraction from vulnerability detection. While IPEA leverages software memory tagging for security, its instrumentation-heavy approach results in significant overhead, making it impractical for deployment. In contrast, RemOTA introduces a checkpoint-based attestation mechanism to reduce runtime costs while targeting temporal bugs. Emerging research has also explored pointer integrity in embedded and real-time systems. For example, Partial Context-Sensitive Pointer Integrity for Real-time Embedded Systems (Wang et al., 2024) and Control-flow Integrity for Real-time Embedded Systems (Moghadam et al., 2021) investigate pointer and control-flow guarantees under timing constraints. RemOTA builds upon these foundations but uniquely focuses on detecting temporal memory errors using minimal on-device instrumentation and offloaded memory emulation.

Temporal memory errors, particularly Use-After-Free. Temporal memory safety remains one of the most dangerous classes of vulnerabilities. In desktop, kernel, and server environments, extensive research has led to sophisticated detection techniques. Tools like FreeWill (He et al., 2022), FFmalloc (Wickman et al., 2021), RTT-UAF (Du et al., 2024), UAFX (Zhang et al., 2025), and DangZero (Gorter et al., 2022) have demonstrated strong detection capabilities by instrumenting memory operations, leveraging shadow memory, or re-engineering memory allocators. However, these techniques come at a high cost in terms of runtime overhead, memory usage, and reliance on system-level features (e.g., virtual memory, page tables), making them fundamentally unsuitable for low-power, resource-constrained microcontrollers.

To illustrate this mismatch: DangZero detects UAFs by accessing page table metadata to trace memory usage efficiently, a technique impossible to implement on bare-metal MCUs without an MMU. FFmalloc, on the other hand, introduces a custom allocator that enforces a one-time allocation policy to ensure freed memory is never reused, thereby preventing UAFs entirely. While effective, this level of allocator control is impractical in embedded systems where developers often rely on lightweight or custom heap implementations optimized for real-time constraints.

Moreover RemOTA addresses a more critical gap. Drawing inspiration from FFmalloc and DieHard (Berger and Zorn, 2006), RemOTA integrates a one-time allocation emulation into a remote verifier, avoiding any need to modify the allocator on-device. Instead of tracking every memory access or maintaining shadow memory structures, RemOTA shifts the complexity off the device.

Loosely Synchronized and Whole-Program Attestation. RemOTA also relates to recent work on control-flow analysis, such as Blast (Yadav and Ganapathy, 2023b), and PittyPat Ding et al. (2017), which adopt path-sensitive or loosely synchronized attestation techniques. While these systems offer high fidelity for control flow checking, RemOTA extends this model by incorporating data-flow emulation to detect *temporal* safety violations, enabling the system to detect use-after-free errors with limited checkpoints instead of full instruction logging.

Sanitizers. Sanitizers help detect bugs such as memory errors, concurrency issues, undefined behavior, and type confusion (Haller et al., 2016). For memory errors, two main approaches exist: redzone-based and pointer-based. Redzone methods (Serebryany et al., 2012; Salehi et al., 2020; Yong and Horwitz, 2003) surround objects with invalid regions and use shadow memory to catch out-of-bounds accesses. Pointer-based methods (Nagarakatte et al., 2009a; Necula et al., 2005; Nagarakatte et al., 2009b; Austin et al., 1994; Duck et al., 2017) encode access bounds in pointers, either as fat pointers (Necula et al., 2005) or via external metadata (Nagarakatte et al., 2009b, 2010). While tools like ASan effectively detect memory bugs, they incur high memory and performance overheads (Jeon et al., 2020). Recent efforts reduce overhead through optimized metadata (Jeon et al., 2020; Ba et al., 2023) or hardware support (Li et al., 2022; Zhang et al., 2019). However, most sanitizers are testing-oriented and overlook temporal properties.

3. Background

UAF Detection. Tracking each memory allocation and deallocation operation, such as `malloc()`, `calloc()`, and `free()`, is crucial to detect use-after-free vulnerabilities through dynamic analysis. This approach precisely pinpoints when a memory block is valid or invalid. Beyond watching these allocation events, it is crucial to monitor every pointer access within an object's boundaries to catch any misuse. Static analysis can help by pinpointing pointers that definitely do not reference any memory blocks, effectively narrowing down the possible pointers that might access a specific block. This assists in optimizing instrumentation by limiting checks to pointer accesses that could potentially alias the originally allocated object rather than every single pointer. Unfortunately, static pointer-alias analysis tends to be impractical, usually yielding less accurate and broader alias resolutions.

Remote Attestation. This widely used method ensures the integrity of the execution of resource-limited embedded systems. Using an external trusted entity, called the verifier, remotely evaluates the condition of a target device or application. For integrity assurance, the prover usually integrates a trust anchor, such as ARM TrustZone, into embedded devices to protect the authenticity of the collected execution traces. These traces compile the attestation report, periodically sent to the verifier for confirmation, serving as a mechanism to verify runtime integrity in embedded or IoT devices.

ARM TrustZone M for embedded devices. ARM TrustZone M is a security feature available on select Armv8-M processors, as well as on A-profile cores. TrustZone introduces a distinct security level that operates alongside the processor's existing execution modes, dividing the system into a secure (S) world and a nonsecure (NS) world. Both worlds support defined memory regions; peripherals and flash can be allocated exclusively to one world. On TrustZone-M processors, the security mechanism is memory-mapped based, and the secure world can have multiple entry points located in a specific region called the nonsecure callable region.

Microcontrollers. In contrast to the microprocessors found in personal computers and smartphones, microcontroller units (MCUs) are engineered for much lower power usage, function at reduced clock speeds (generally less than 500 MHz) and incorporate built-in SRAM and flash memory, which typically do not exceed a few hundred kilobytes. The programs that operate on MCUs, known as firmware, are tightly linked to the hardware that they control. Because of these resource limitations, devices built on MCUs frequently miss conventional components such as a memory management unit (MMU) that are standard on PCs. An identifying feature of MCU firmware is its operation within a unified flat address space that integrates components such as the kernel (if included), user tasks, and memory-mapped peripherals. Methods for bug detection typically depend significantly on the runtime environment of the system under examination.

4. Threat model

We assume the firmware code instrumentation and the compiler utilized to produce the firmware are correct and uncompromised. When the system is running, we model an adversary capable of delivering arbitrary payloads or inputs to the device during firmware execution, potentially exploiting temporal memory errors. Specifically, for RemOTA, we focus on Non-secure World attacks, such as Use-After-Free vulnerabilities (Szekeres et al., 2013; Lee et al., 2015), which can lead to severe outcomes such as data leakage from firmware applications (Costin et al., 2014), exposing critical information within embedded software. These attacks may also result in program crashes (Chen et al., 2018) or trigger abnormal behavior. The attacker might try to interfere with the data collected by the instrumentation, but such attempts are thwarted by employing ARM TrustZone, which safeguards the integrity and confidentiality of sensitive operations. Hardware attacks on the device and TrustZone, including side-channel analysis, fault injection, cold boot extraction, and other similar methods, are beyond the scope of this work. RemOTA assumes the presence of a hardware-enforced trust anchor, such as ARM TrustZone-M to guarantee the integrity and authenticity of execution traces. Misconfiguration or absence of such a trusted execution environment is considered out of scope, consistently with prior remote attestation systems (e.g., C-FLAT, OAT, BLAST). In the absence of TrustZone (or an equivalent TEE), RemOTA degrades to a monitoring-only mechanism suitable for debugging or testing, but does not provide security guarantees against adversaries capable of tampering with execution traces.

It is important to note that RemOTA effectively detects Use-After-Free vulnerabilities but cannot catch control-flow hijacking or microarchitectural attacks that bypass selective instrumentation. We remark that RemOTA's design centers on tracking the firmware heap object to identify Use-After-Free attacks. Conversely, modeling the correct execution flow and detecting spatial errors is an orthogonal problem already addressed by state-of-the-art runtime remote attestation protocols, such as OAT (Sun et al., 2020), C-FLAT (Abera et al., 2016), or Blast (Yadav and Ganapathy, 2023a), which remain vulnerable to UAF attacks and microarchitectural attacks as well.

5. RemOTA system overview

We provide an in-depth discussion of the system architecture, detailing its components and operational workflow (Section 5.1); to illustrate the capabilities of RemOTA, we present a practical example of vulnerability analysis in embedded systems (Section 5.2). Subsequently, we describe the deployment scenario and discuss the application case (Section 5.3).

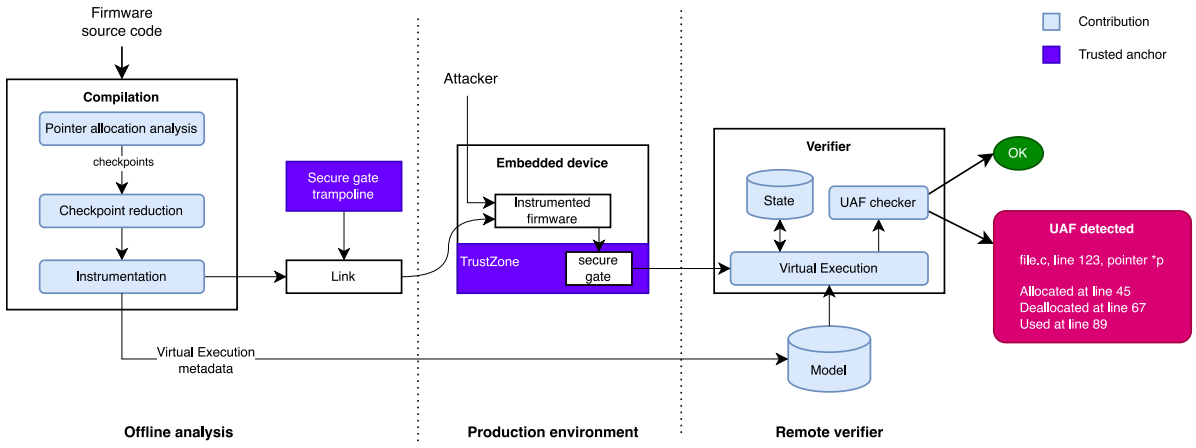

Fig. 1. REMOTA architecture. During Offline Preparation, the firmware source code is compiled and analyzed. In this phase, we obtain an instrumented firmware and a model that marks all checkpoints, use, allocation and deallocation functions. During the verification, the instrumented firmware is executed on an embedded device and reports a trace T to a Verifier. The latter validates the trace T against the model M.

5.1. System architecture

The system architecture depicted in Fig. 1 consists of two main components: *offline analysis* and *online virtual execution*.

During the *offline analysis*, we analyze the source code to: (i) **identify instrumentation points**, allowing the firmware to report operations during execution, and (ii) **create a data model** to extract metadata from the firmware. Identifying instrumentation points involves static analysis to locate key execution paths, function entry/exit, and memory operations. The benefit of our static analysis is to reduce the instrumentation points needed to trace the firmware code while reconstructing its memory state in the Verifier. The model includes insights such as control flow graphs and memory allocation patterns to support an accurate and efficient verification process.

During the *online virtual execution*, the firmware runs on the board, transmitting each operation to the verifier in real time. *The verifier operates in parallel, independently of the firmware's execution*, and replicates these operations to monitor the lifecycle of memory objects using metadata from the offline analysis while maintaining an optimized representation of the heap objects state. Efficient data structures track memory allocations and deallocations with minimal latency. Advanced caching mechanisms reduce the firmware's interaction overhead with the verifier by aggregating multiple checkpoints into a single chunk. These mechanisms (Section 6.4) also ensure a balance between integrity, confidentiality, and performance of the checkpoint metadata by leveraging ARM TrustZone. Sensitive information is stored in a TrustZone secure cache to prevent data breaches. For security, all communication channels are also protected by ARM TrustZone, and the verifier authenticates the firmware's traces, ensuring data integrity.

5.2. Running example

```
1  void vulnerability(){
2      a = malloc(100);
3      c = a;
4      free(a);
5      b = malloc(20);
6      use(c);
7      if(rand() % 2){
8          c = b;
9      }
10     use(c);
11 }
```

Listing 1: Running example with aliasing problem

Listing 1 illustrates a running example that demonstrates the challenges associated with UAF detection in resource-constrained environments and presents the core ideas of REMOTA. The example starts with a flawed code snippet featuring two Use-After-Free vulnerabilities. The first UAF is straightforward and occurs at `line 6`, where the variable `c` (alias of the variable `a`) is used after being freed at `line 4`. The second UAF, involving the pointer `c`, occurs at `line 10` if the branch at `line 7` is not taken. The offline analysis identifies two *allocation points*, that correspond to `malloc` calls and two *checkpoints*, that correspond to pointer-related flow changes. The checkpoints are computed based on CFG and are the post-dominator nodes of pointer operations, that is, the nodes that dominate any pointer access along an uninterrupted path. Such checkpoints are able to capture multiple uses along the same path without instrumenting all the pointers inside. With such checkpoints, the REMOTA verifier is able to select (by using the checkpoint ID information) the unique dynamic path that the program is executing. In our example, the checkpoints are one to track whether the branch at `line 7` is taken and another to confirm the function return. More in detail, the offline analysis generates the following instrumentation points:

- `Alloc 2 100` (line 2, alloc of size 100 byte)
- `Alloc 5 20` (line 5, alloc of size 20 byte)
- `CP 9` (line 9, the program has reached line 9)
- `CP 11` (line 11, the program has reached line 11)

With such instrumentation points, the analysis also derives the following execution paths:

- **Path A**: `line 2,3,4,5,6,7,8,10`
- **Path B**: `line 2,3,4,5,6,7,10`

The verifier can receive one of the following traces:

- T_1: [`Alloc 2 100`, `Alloc 5 20`, `CP 9`, `CP 11`]
- T_2: [`Alloc 2 100`, `Alloc 5 20`, `CP 11`]

In this context, **Path A** contains two use-after-free (UAF) vulnerabilities, while **Path B** contains only one. During firmware execution, a series of traces is received by the verifier. Each trace is composed of a list of information that can be of two types: (1) the ID and size of a memory allocation operation, and (2) the ID of a checkpoint. Based on this information, the verifier determines which execution path to emulate. The verifier processes these inputs as follows. When an allocation operation is received, it extracts the allocation size from the trace and creates a new virtual memory chunk in the format: `CHUNK ⟨id,`

`starting_address, ending_address`⟩. When a checkpoint is received, the verifier uses the checkpoint ID to select the corresponding execution path and emulates all pointer-related instructions along that path.

For example, consider a trace T_1, where the verifier first receives two allocation operations: `a = malloc(100)` and `b = malloc(20)`. Upon receiving the first checkpoint (`CP 9`), the verifier begins emulating the instructions starting from `line 2` up to `CP 9` (line 9). This sequence includes operations such as: `a = malloc(100)`, `c = a`, `free(a)`, ..., `c = b`. During emulation, each allocation is handled by the *Virtual Allocator (VA)*, which emulate the memory allocation by creating a fresh memory block, adds it to the chunk table, and returns the starting address, which is then stored in the pointer table. At any point in time, the virtual memory state of the firmware, reconstructed remotely, is defined by the current state of the *chunk table* and the *pointer table*.

Given the following architecture to detect Use-After-Free vulnerabilities, REMOTA uses a one-time allocator OTA (Wickman et al., 2021) as a virtual allocator, which guarantees the sequential virtual allocation of new chunks without re-using released memory. The use of an OTA in combination with the framework design is the key concept to verify whether a pointer's usage involves an outdated memory chunk that has already been freed. Specifically, when a `free` operation is executed, the corresponding fragment entry is tagged as "freed" and any pointer associated with the deallocated fragment is no longer valid. If the freed pointer is used later by a memory operation, a UAF alert is generated. For instance, when the verifier processes `line 6` (`use(c)`), it checks the status of `c`. As `c` indicates a freed fragment, the verifier raises a UAF alert. However, if the execution proceeds to `CP 11`, the terminal operation (`use(c)`) does not prompt another UAF due to the preceding `line 8`, which modifies `c` to refer to a valid active chunk.

The use of OTA is crucial because the verifier is able to identify if a pointer is referring to a freed chunk; if a classic allocation method is used, it is possible to reallocate over freed chunks, but with "chunk reuse" the verifier is not able to distinguish if the pointer is pointing to the old dead chunk or the new alive one. Since REMOTA employs an OTA allocator, every new chunk is allocated sequentially without reuse, and this is useful to spot which chunk the pointer refers to.

5.3. Application & Deployment scenario

In a typical deployment, an embedded device running REMOTA participates in a continuous, asynchronous remote-attestation workflow. As the firmware executes on resource-constrained hardware, it non-intrusively reports to the verifier two types of events: allocation markers (`Alloc`) and control-flow checkpoints (`CP`). Such traces incur a minimal blocking overhead: after queuing each event, the firmware immediately resumes its primary tasks (sensor sampling, actuation, communication, etc.). When the cache is full, then the events chunk is transferred to the verifier without waiting for network transfers or verification results. By decoupling detection from prevention and running firmware logic in tandem with verification, REMOTA achieves continuous firmware monitoring aligned with state-of-the-art attestation systems (OAT, C-FLAT, Blast) while preserving the device's real-time responsiveness. Within this architecture, attacks that exploit temporal memory errors such as Use-After-Free in the Non-Secure World are flagged by the verifier soon after they occur, Section 7.4, enabling timely alerts or remediation steps on the management plane. Sensitive attestation packets are protected end-to-end by TrustZone's secure channel, and the verifier's trusted status ensures that even if an adversary delivers malformed payloads at runtime, any deviation from the expected CP-flow is detected without disrupting the device's primary function. Upon detecting a Use-After-Free event, REMOTA's verifier can coordinate a secure response: the system supports remote resetting via TrustZone-enabled frameworks such as Ninja (Ning and Zhang, 2017) for secure debugging. In practice, the verifier issues an authenticated reset command that halts the compromised firmware, purges any corrupted state, and reboots the device into a known-good configuration.

6. REMOTA framework

A major challenge in firmware analysis is the impactful balance between minimizing overhead and accurately identifying program variables utilized during the firmware execution phase. Achieving this requires a careful approach to tracking dynamically allocated data, such as active objects, while efficiently handling memory operations. Our system addresses this challenge by monitoring memory allocation events and virtually executing allocation operations to capture relevant runtime data without introducing excessive performance penalties. In the following sections, we provide a detailed overview of how REMOTA achieves effective instrumentation through the definition of an allocation data model (Section 6.1). This model plays a crucial role in identifying key instrumentation points necessary for precise runtime tracking. We then introduce our execution protocol (Section 6.2), which demonstrates how our system reconstructs the execution flow of the firmware, ensuring comprehensive coverage and accurate tracking of memory-related operations. Furthermore, we explain how REMOTA enforces rigorous checks to detect and prevent use-after-free (UAF) vulnerabilities, thus improving firmware security and reliability. Finally, we discuss the optimizations employed by REMOTA to improve the device's performance (Section 6.4).

6.1. Allocation data model

This section presents our strategy for identifying Checkpoints (CPs) within the firmware and extracting the memory allocation model. A naive approach would instrument all pointer reads and writes during execution. However, this leads to an excessive number of instrumentation points and significantly degrades device performance. Preliminary tests show that instrumenting every memory operation results in a 4.29× increase in instrumentation points compared to our optimized method (Section 7). To reduce this overhead, we exploit the fact that memory operations within basic blocks execute in a deterministic order. Instead of instrumenting every memory access, we identify key control-flow points where execution paths may diverge. These control-flow divergence points are detected using post-dominator analysis on the program's Control Flow Graph (CFG). While post-dominator nodes are traditionally used to locate control-flow convergence points, we reinterpret them to detect execution forks. Specifically, we identify nodes whose immediate post-dominator lies outside their control region, indicating a branching point in the execution. By placing checkpoints at such points, we segment the firmware into logical execution paths where memory operations can be grouped and reasoned about precisely. This approach significantly reduces instrumentation while preserving the verifier's ability to emulate memory behavior and verify execution integrity. Our algorithm identifies two primary types of node points in CFG:

- `Alloc` points, representing dynamic memory allocation sites (e.g., `malloc`, `calloc`, `realloc`).
- `Use` points, representing any instruction that accesses a memory object allocated on the heap, free functions included.

Allocation points are identified by wrapping standard allocation functions using our heap detection library. Use points are identified through the construction of a *Value Flow Graph* (VFG), which models how values propagate across the program. To extract precise *def-use* chains, we rely on Static Single Assignment (SSA) form during compilation. Our framework traverses the VFG to associate each `Alloc` node with its corresponding `Uses`. The VFG is built atop an *Interprocedural Control Flow Graph* (ICFG), which captures control flow across function boundaries. To resolve indirect function calls, we use Andersen's pointer analysis (Sui and Xue, 2016). As with any static points-to analysis, this step is inherently approximate and may omit feasible pointer-dependent paths due to unresolved aliasing. Importantly, such

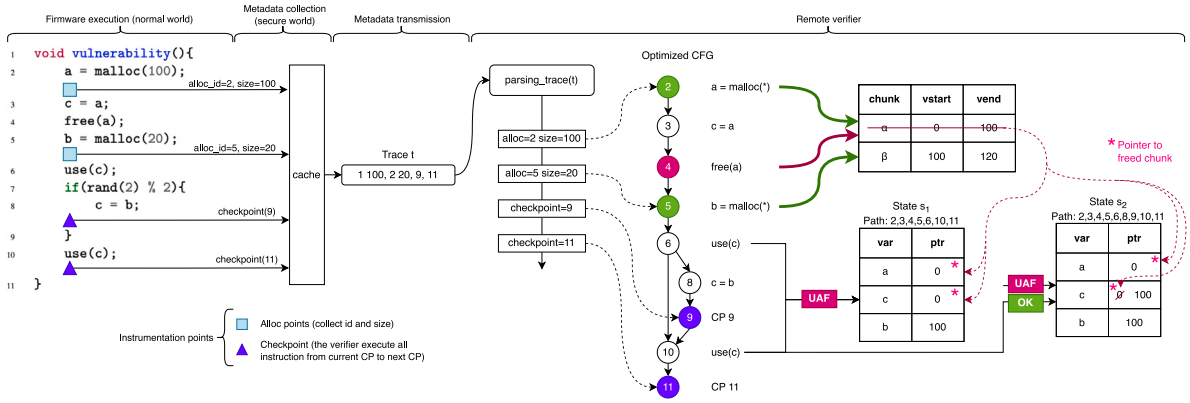

Fig. 2. Overview of REMOTA: The system instruments the program to produce execution traces containing checkpoints and allocation events. These traces are sent to the verifier, which parses them, maintains a graph model of the program's control flow and performs virtual execution to update its internal state and construct the allocation table. Then the verifier checks for use-after-free in every virtual execution.

imprecision affects coverage but not correctness. In REMOTA, static analysis is used exclusively to guide checkpoint placement and identify candidate execution regions, and is never used to decide whether a pointer dereference is safe. All alias resolution and Use-After-Free detection are performed dynamically by the remote verifier through virtual execution between checkpoints. Consequently, the system precision refers to the absence of false positives: every UAF reported by REMOTA corresponds to a concrete dereference of a memory chunk explicitly marked as freed in the reconstructed heap state. After the VFG is constructed, we perform post-dominator analysis to identify the divergence points mentioned earlier. We use a standard iterative data-flow algorithm: each node initially assumes all nodes as its post-dominator set, and the sets are refined through intersection with the sets of successor nodes until convergence. From this, we compute the immediate post-dominator for each node. While these typically indicate where paths reconverge, we invert this information using the relationship to locate where control flow originally diverged. These locations serve as optimal points for checkpoint insertion. Our analysis finally produces the following instrumentation points:

- `Checkpoint (CP)`: Marks a control-flow divergence in the program, indicating that the firmware has reached this *CP* and summarizes all *use* operations from the previous *CP* to the current one.

By limiting instrumentation to these split points, we maintain the accuracy of memory tracking while significantly reducing runtime overhead.

6.2. Execution protocol

In this section, we present the protocol of REMOTA, focusing on the dynamic component. We begin by detailing the reconstruction of execution flow from checkpoints, and finally, we describe the method used for detecting use-after-free vulnerabilities.

Flow reconstruction and virtual execution. The static analysis described above enables the detection of potential use-after-free (UAF) vulnerabilities by tracking the lifecycle of memory objects—specifically their allocation, usage, and deallocation. A central element of this process is points-to analysis (PTA), which aims to determine the set of memory locations (or objects) that a pointer variable may reference at various points in the program. This information is essential for understanding how memory is accessed. If PTA reveals that a pointer may reference a memory object that has already been deallocated (freed), then a use-after-free condition may exist. By mapping pointer references to their possible target objects, PTA helps determine whether any pointer usage could lead to invalid memory access—an essential step in statically identifying UAF bugs. A fundamental challenge arises from the inherent imprecision of PTA. Most PTA algorithms used in modern static analysis frameworks (e.g., Andersen's Andersen (1994) or Steensgaard's analysis Steensgaard (1996) and Hind and Pioli (2000)) tend to produce over-approximated results for aliasing pointers. Therefore, any static analysis cannot conclusively determine whether a pointer `Use` is safe or potentially dangling, which undermines the effectiveness of static UAF detection. This limitation motivates our decision in REMOTA to offload precise alias resolution to a remote verifier via dynamic emulation, rather than relying solely on static methods.

The scenario depicted in the Listing 1 highlights this problem: along the path to a `Use`, there may be several pointer assignments that depend on runtime conditions, making it infeasible to determine statically what the `Use` refers to. Consequently, the PTA produces conservative results that indicate variables `c` and `b` *may* point to the same location. Due to this constraint, we opted to remove points-to analysis used for solving the pointed objects and instead implement virtual execution. We outsourced the verifier to a remote virtual executor, which functions by receiving checkpoints from the firmware and virtually carrying out instructions from the current position to the one specified by the received checkpoint. Receiving a checkpoint indicates that the firmware has reached that specific segment of the code. As described in REMOTA architecture 2, the verifier obtains checkpoints transmitted from the embedded device via the chosen communication channel. Upon receiving the trace, the verifier processes the data, reconstructing the sequence of instructions to be executed using the model. Subsequently, the instructions are executed in a virtual environment, with the virtual state updated after each instruction and inspected for possible use-after-free vulnerabilities. The verifier records the memory state of firmware execution using three primary tables (Fig. 2): (i) **a table of IR registers** (e.g., `%c = add ptr %a, %b`, where %a, %b, %c represent registers), (ii) **a table for global variables and stored memory pointers**, and (iii) **a table for allocated memory segments** containing temporary values. The execution model REMOTA predominantly takes into account the memory segments allocated during run-time, the pointers that regulate these segments, and the variables affecting memory access. We define Virtual Instruction (VI) any LLVM IR instruction that involves these pointers (Section 6.3).

6.3. Virtual execution instruction list

Table 1 lists the virtual execution instructions that the verifier can emulate to model the firmware's behavior and maintain the memory object allocation. Each LLVM instruction is listed with its parameters and a representative LLVM IR example. To effectively model the

Table 1
LLVM instructions supported by the verifier with parameters and examples.

Instruction	Parameters	Example (LLVM IR)
`store`	Value to store, destination pointer	`store i32 42, i32* %ptr`
`load`	Source pointer	`%val = load i32, i32* %ptr`
`getelementptr`	Base pointer, indices, type info	`%gep = getelementptr i32, i32* %ptr, i32 3`
`call`	Function name, arguments	`%res = call i32 @foo(i32 %a, i32 %b)`
`bitcast`	Source value, destination type	`%cast = bitcast i8* %ptr to i32*`
`icmp`	Comparison type, operands	`%cmp = icmp eq i32 %a, %b`
`sext`	Source value, destination type	`%ext = sext i8%val to i32`

memory object allocation during firmware execution, it is not necessary to emulate the full set of LLVM instructions. Instead, RemOTA focuses on a carefully selected subset that captures relevant pointer operations and memory interactions. Instructions such as `load`, `store`, and `getelementptr` directly reflect memory access and pointer arithmetic. Type transformations (`bitcast`, `sext`) and comparisons (`icmp`) provide the necessary context for understanding how pointers are used and interpreted. Function calls (`call`) captures interprocedural behavior and memory allocation/deallocation functions. By emulating such instruction subset, the verifier can reconstruct the structure and lifetime of heap objects with sufficient accuracy for our dataset, while avoiding the overhead of simulating unrelated instruction.

Trace validation. The verifier is one of the main contributions of RemOTA, it keeps the allocation map (start and end of chunks) and virtually executes the LLVM instructions by using the graph model extracted in the offline phase as a new trace arrives.

When a trace is received (Algorithm 6.1), the verifier updates the allocation table that comprises the remote state of the firmware and processes the instruction using virtual addresses. For an *Alloc* operation (e.g. `Alloc 24 100`), the verifier creates a new entry in the allocation table, assigning a virtual chunk through an allocator (for detecting UAF we use a one-time allocator strategy). The chunk is marked as *alive*, with its starting address determined by the OTA and its end address calculated as the start address plus the specified size. Upon receiving a *checkpoint* operation (e.g. `CP 8`), the verifier virtually executes all intermediate *Use* operations, including allocations, deallocations, and pointer manipulations, until reaching the designated checkpoint. For a *free* operation, the corresponding virtual fragment is flagged as *not alive*. Furthermore, when the verifier processes a *Use* operation, it also performs a use-after-free check by verifying that the object being accessed is marked as *live*. If the object is not alive, an alarm is raised; otherwise, the virtual instruction proceeds as intended.

One-time allocator strategy. To simulate the memory allocation initiated by the firmware, the verifier must adopt an allocation strategy. We have chosen a one-time allocator (OTA) approach, which assigns unique virtual addresses for each new allocation and does not reuse memory. To justify our choice of allocation strategy, consider the example in Listing 1: After executing `line 4`, allocation `a` becomes invalid. The `use(c)` operation in `line 6` is thus invalid, as it refers to freed memory. In this scenario, the validity of `use(c)` in `line 10` depends on whether the conditional branch was taken, reassigning `c` to point to `b`. The scenario in `line 10` highlights a significant challenge: when a new pointer (`b`) reuses the memory address of a previous allocation (`a`), the verifier cannot determine whether `use(c)` refers to valid or freed memory. With point-to static analysis, the verifier is unable to distinguish if `use(c)` on `line 10` references `a` or `b`, since pointer assignments on `lines 3` and `8` introduce aliasing; with `line 8` executing conditionally.

Even with access to dynamic address information, the verifier remains unable to resolve the issue due to the firmware's memory allocation strategy, which allows freed chunks to be reused. This means that a new allocation could potentially overlap with a previous one. Under the classical memory reuse strategy (Table 2), a pointer that has been freed can still be considered valid due to chunk reassignment.

In contrast, the virtual OTA employed by the verifier assigns a unique virtual address to each allocation, effectively eliminating aliasing ambiguities. For example, before `line 8`, the allocation table could resemble Table 3, where `a` and `b` hold distinct virtual addresses. Consequently, a `Use` operation targeting addresses `100` to `104` on `line 8` can be unequivocally associated with `b`, while an operation on addresses `0` to `4` can be linked to `a`. This approach eliminates the

Algorithm 6.1: Trace verification

```
Input: GraphModel GM
Input: Trace T
Input: State S
Function check_uaf(S, addr)
    if not S.chunk.get(addr).alive then
        // uaf detected;
    end if
Function virtual_execution(instruction, S)
    switch instruction.type do
        case Load do
            check_uaf(S, S.memory(S.regs.get(param0)));
            S.regs.add(instruction.label,
              S.memory(S.regs.get(param0)));
        end case
        case Store do
            check_uaf(S, S.memory(param1));
            S.memory(param1) ← param0;
        end case
        case Alloc do
            addr ← S.OTA.alloc();
            S.chunk.add(addr, param0);
            S.regs.add(instruction, addr);
        end case
        case Free do
            S.chunk.get(param0).alive ← false;
        end case
        ...
    end switch
events ← parse_trace(T);
current_cp ← entry_checkpoint;
for e in events do
    target_cp ← e;
    instructions ← unroll_checkpoint(GM, current_cp, e);
    /* Checkpoints are totally ordered along a trace;
      unroll_checkpoint function deterministically returns the
      instruction sequence between cp_from and cp_to. */
    for i in instruction do
        virtual_execution(i, S);
    end for
    current_cp ← target_cp;
end for
```

Table 2
Memory allocation table with classical memory reuse strategy at the end of the snippet. Reference snippet 1.

Alloc	Start	End	Size	Alive
a	0	100	100	No
b	0	20	20	Yes

Table 3
Memory allocation table with OTA strategy at the end of the snippet. Reference snippet 1.

Alloc	Start	End	Size	Alive
a	0	100	100	No
b	100	120	20	Yes

need for point-to analysis, enhancing the verifier's effectiveness and reliability.

6.4. Dual-level cache optimization

This section demonstrates a caching strategy designed to enhance the performance of REMOTA by minimizing the number of calls between the (NS) and the Secure region, which constitutes a significant communication overhead for the checkpoint. In ARM Cortex-M33 processors, secure gate calling follows a specific execution flow (Slamaris, 0000; Microelectronics, 2020; Arm Ltd, 2016). When operating in a nonsecure (NS) region, the processor functions at the non-secure security level. Direct calls from the NS world to the secure (S) world are prohibited. Instead, interactions must go through a designated entry point within a nonsecure callable (NSC) region. Once this entry point is invoked, the processor transitions to the secure state and redirects execution to the corresponding function within the secure world. To enhance the efficiency of communication between the NS and S environments, we employ a batch processing approach to transmit trace data to the verifier. This requires temporarily caching sensitive checkpoint data in the secure world. However, invoking secure functions for each checkpoint individually would result in excessive performance degradation due to the overhead of frequent NS-to-S transitions. To mitigate this issue, we introduce a double-layer caching mechanism that minimizes context switching between the two execution domains. Initially, checkpoint IDs are stored in a first-level NS cache. Once this cache reaches capacity, its contents are transferred to a second-level secure cache via the secure gate. When the secure cache is full, the aggregated data is transmitted to the remote verifier. This hierarchical caching strategy significantly reduces the frequency of NS-to-S transitions, thereby enhancing system performance and reducing processing overhead.

7. Evaluation

We evaluated the properties of REMOTA in three main research questions: **(RQ1)** Does REMOTA correctly detect the vulnerability (Effectiveness Analysis, False positive, and False negative)? **(RQ2)** Can REMOTA be deployed on real platforms (Performance Evaluation)? **(RQ3)** What are the security properties REMOTA?

7.1. Prototype details

All experiments were carried out on an ARM STM32 NUCLEO-L552ZE-Q board, which has ARM TrustZone M support. This board is an ultra-low-power microcontroller based on the high-performance Cortex-M33 32-bit RISC core, equipped with 256KiB of RAM and 512KiB of Flash memory, operating at a frequency of 110 MHz. The STM32 NUCLEO-L552ZE-Q was selected as our reference device for prototyping due to its widespread adoption in commercial applications, offering a balance of performance, power efficiency, and security features. The inclusion of TrustZone M (L5) provides hardware-enforced isolation, allowing the development and evaluation of secure embedded applications. The board's capabilities make it an ideal platform for testing and validating our approach in a real-world embedded environment, allowing us to explore the trade-offs between security and performance under constrained resource conditions. Furthermore, its extensive documentation and strong community support facilitate rapid prototyping and debugging. For offline analysis, graph generation, and compilation, we utilized LLVM version 13, which provides a robust framework for program analysis and transformation. To further enhance our analysis capabilities, we extended the SVF framework, specifically the SVF-2.9 analysis algorithm, by integrating it with the REMOTA Pass.

7.2. Dataset

To evaluate both the accuracy and practical applicability of REMOTA, we employed two complementary datasets: the Juliet test suite and the BEEBS benchmark suite.

The **Juliet test suite** (NIST, 2017) is a widely used collection of synthetic C/C++ test cases designed to systematically cover a broad spectrum of Common Weakness Enumerations (CWEs). It provides fine-grained control over vulnerability injection and is particularly useful for benchmarking detection accuracy and comparing different analysis tools under controlled conditions. For our purposes, we focused on the CWE-416 category, which includes test cases explicitly designed to trigger Use-After-Free (UAF) vulnerabilities (126 use cases). Since REMOTA operates at the LLVM IR level, we were able to run these cases directly without requiring architecture-specific modifications. While the Juliet test suite offers an exhaustive and structured set of UAF cases, it does not fully reflect the execution environment or programming constraints of extremely low-power, low-resource microcontrollers. In particular, it abstracts away hardware-level limitations such as the absence of an MMU and the use of flat memory models, factors that directly affect memory allocation strategies in embedded systems. Nevertheless, the core semantics of UAF vulnerabilities, namely, the dereferencing of freed pointers, are consistent across both desktop and embedded platforms. We remark that Juliet is not intended to model realistic firmware complexity, but to provide a controlled ground truth for CWE-416 vulnerabilities, enabling unambiguous measurement of detection precision. The use of both datasets together allow us to evaluate both correctness under known ground truth and deployability under realistic firmware workloads.

To complement Juliet with real-world embedded code, we also used the **BEEBS** (Pallister et al., 2013), which has been adopted in prior works such as BLAST (Yadav and Ganapathy, 2023a) and IPEA (Shi et al., 2024). BEEBS is specifically tailored for embedded systems and contains a variety of benchmarks representative of realistic firmware applications. In particular, the dataset comprises 15 test firmware tailored for embedded systems. Firmware refers to background system code that implements device functionality autonomously, typically without requiring direct user interaction or input. It is designed to operate continuously or reactively in response to internal events, hardware triggers, or scheduled tasks, rather than explicit user commands. Unlike Juliet, BEEBS exposes memory usage patterns that arise in actual embedded deployments, including both static and dynamic memory allocations. It is often assumed that embedded firmware relies exclusively on static allocation; however, BEEBS includes multiple examples where dynamic memory is used—either because it is necessary due to runtime variability, or because it simplifies resource management in specific applications. These cases provide strong empirical evidence that dynamic memory management is not uncommon in embedded code, thereby reinforcing the practical relevance of REMOTA's detection capabilities. Moreover, BEEBS benchmarks exhibit non-trivial control-flow complexity, including nested loops, conditional branches, and function calls that reflect real-world firmware logic. For example, across the BEEBS suite, benchmarks contain an average of over 52 conditional branches, with

Table 4
Performance overhead of REMOTA. The table reports the baseline execution time (without instrumentation), along with both static and dynamic checkpoint data. Specifically, *Static CP* refers to the number of checkpoint instructions inserted into the firmware during offline analysis, while *Dynamic CP* denotes the total number of checkpoint triggers observed during execution. We also include the execution time for the fully instrumented configuration, the execution time with REMOTA, and the total size of the generated trace data.

Firmware	Baseline	Fully instrumented				RemOTA			
	time (s)	CP	Dynamic CP	Trace (byte)	Time (s)	CP	Dynamic CP	Trace (byte)	Time (s)
aha_mont64	31	35	144 432	289 748	72 (+56.94%)	33	137 392	275 668	70 (+55.71%)
st	4 670	24	1 593 022	3 197 484	5097 (+8.38%)	22	1 590 162	3 191 764	5097 (+8.38%)
edn	12 065	32	40 911 752	81 823 512	23 109 (+47.79%)	16	3 081 542	6 163 092	12 906 (+6.52%)
fir	73 340	14	289 245 002	578 534 008	152 301 (+51.85%)	8	15 422 002	30 888 008	77 444 (+5.30%)
nettle-aes	8 092	51	8 700 122	17 400 256	11 044 (+26.73%)	40	7 001 282	14 002 576	10 431 (+22.42%)
libhuffbench	8 454	78	66 026 072	132 056 988	26 529 (+68.13%)	20	17 806 362	35 617 568	13 299 (+36.43%)
ud	10 343	32	47 961 102	95 922 204	23 274 (+55.56%)	2	1	2	10 333 (−0.10%)
nbody	3 131	18	685 082	1 370 172	3309 (+5.38%)	8	11 222	22 452	3133 (+0.06%)
minver	6 161	57	14 285 702	28 571 408	10 344 (+40.44%)	4	61 052	122 108	6177 (+0.26%)
matmult-inc	13 558	21	44 740 522	89 481 048	25 083 (+45.95%)	4	5062	10 128	13 560 (+0.01%)
tarfind	5 464	22	10 046 298	20 113 276	8278 (+33.99%)	22	10 046 298	20 113 276	8278 (+33.99%)
nettle-sha256	12	35	7372	15 184	14 (+14.29%)	10	1212	2864	13 (+7.69%)
sglib-combined	10 232	657	98 054 222	196 108 444	39 330 (+73.98%)	2	1	2	10 218 (−0.14%)
primecount	10 245	18	96 827 062	193 655 008	42 490 (+75.89%)	14	96 436 342	192 873 568	42 379 (+75.83%)
crc32	44	8	112 862	225 728	73 (+39.73%)	4	112	22	44 (+0%)
Arithmetic mean		**33.75**	**47 767 703.3**	**95 542 341**	–	**16.75**	**12 633 327.5**	**25 273 589.33**	–
Geometric mean					**+31.01%**				**+4.47%**

some reaching up to 355. Similarly, memory-related instructions range up to 547 per benchmark, with certain firmware examples exceeding 3000 memory operations, including heap allocations. This level of complexity poses significant challenges for both static and dynamic analysis tools, making BEEBS an ideal dataset for evaluating REMOTA under realistic conditions. By combining Juliet's systematic coverage with BEEBS's realism, we ensure that our evaluation captures both theoretical detection precision and applicability to real-world firmware.

7.3. RQ1:bugs detection

Detection rate

Our evaluation demonstrates that REMOTA successfully detects 100% (126 out of 126 total) CWE-416 Use-After-Free cases in the Juliet dataset.

False positive & False negative

Theoretical false negatives may arise in specific cases where the initial offline instrumentation phase fails to capture the full context required to virtually execute malloc-related operations. These cases typically involve aliasing between pointers and memory objects or limitations in the static analysis algorithm's ability to track pointer usage precisely. Since our algorithm does not directly model such aliasing relationships, these scenarios can lead to missed detections. Regarding the likelihood of such false negatives, we observe that in embedded firmware, indirect calls are typically introduced through a small number of function pointers used for callbacks, interrupt handlers, or driver interfaces. To address this, static-level data back-propagation techniques, such as backward slicing, may be applied. However, when aliasing cannot be resolved statically, a dynamic alias analysis component must be introduced to enrich the verifier's context and allow for broader instruction coverage during emulation. Designing such a component poses significant challenges and is part of our planned future work. In our case the system precision refers to the absence of false positives. While false negatives are theoretically possible due to incomplete static coverage (e.g., unresolved aliasing during checkpoint selection), we did not observe such cases in our evaluation. REMOTA deliberately prioritizes zero false positives and low overhead over full completeness.

This guaranty comes from how our verifier operates: a Use-After-Free (UAF) alert is raised only when there is definitive evidence that a pointer refers to a memory region that has been explicitly freed. Unlike many static tools, our verifier neither speculates about pointer behavior nor assumes worst-case scenarios in the absence of proof. Instead, it relies strictly on concrete evidence (e.g., dynamic emulation techniques). As a result, false positives are structurally impossible by design—both theoretically and as confirmed by our empirical results. In summary, while REMOTA may exhibit false negatives due to inherent static analysis limitations, it ensures zero false positives by construction.

Instrumentation trade-offs

A potential approach to mitigate theoretical false negatives is to instrument all instructions involved in pointer dependencies throughout firmware execution. However, as previously discussed, accurately identifying every such instruction is challenging. To approximate this ideal scenario, we conducted an experiment in which we instrumented all basic blocks containing at least one memory-accessing instruction. This configuration represents the *theoretical upper bound* of our system—maximal instrumentation for complete detection coverage. By emulating every memory-related instruction on the verifier side, this fully instrumented setup ensures that Use-After-Free (UAF) vulnerabilities cannot go undetected, regardless of complex data dependencies or indirect pointer relationships. However, this level of coverage incurs significant costs: our experiments show a *geometric mean runtime overhead of* 31% on the device (Table 4, Fully instrumented). In contrast, our *optimized configuration* uses *selective instrumentation*, targeting only pointer operations directly linked to memory allocations. This approach reduces the average runtime overhead to 4.47% (excluding 0% values from the table), while also achieving a 3.78× reduction in trace data volume—a crucial improvement for embedded systems with limited bandwidth. These results highlight that selective instrumentation provides a practical and efficient trade-off: it maintains a high detection precision with significantly reduced overhead. Moreover, it demonstrates the flexibility of our framework, enabling tunable deployments where users can adjust the balance between detection coverage and system performance based on the specific constraints and criticality of their embedded applications.

Evaluation on real-world embedded vulnerabilities

To evaluate the applicability of RemOTA beyond synthetic benchmarks, we analyze real-world use-after-free (UAF) vulnerabilities in Zephyr RTOS and ESP-IDF. Our goal is to assess whether RemOTA can correctly model and detect vulnerabilities arising in realistic firmware code, rather than to perform a full-system performance evaluation. Instead of instrumenting and executing the entire firmware stacks, we focus on the *vulnerable execution paths*. Specifically, for each vulnerability, we identify and port the chain of functions involved in the allocation–deallocation–use sequence, preserving the original control flow and memory semantics. This enables us to reproduce the vulnerability-triggering behavior in a controlled environment while maintaining fidelity to the original code. We then apply the RemOTA static analysis pipeline to these extracted paths, constructing the corresponding control-flow and value-flow representations and identifying allocation sites, use points, and checkpoint locations. This allows us to verify that the vulnerable code is compatible with our allocation model and that the required instrumentation points can be correctly generated. The resulting traces are processed by the remote verifier, which successfully detects UAF conditions through virtual execution. Using this methodology, we reproduce CVE-2017-14201 and CVE-2025-7403 in Zephyr, and CVE-2026-25507 in ESP-IDF. In all cases, RemOTA successfully detects the resulting UAF conditions, demonstrating that our approach generalizes to real-world firmware patterns. A full-system evaluation on Zephyr or ESP-IDF is non-trivial due to platform and integration constraints. In particular, ESP-IDF targets different architectures (Xtensa/RISC-V), whereas our prototype is implemented on an ARM Cortex-M33 platform with TrustZone-M support. For Zephyr, a complete evaluation would require integrating our LLVM-based instrumentation pipeline into its build system and toolchain, including adapting compilation, linking scripts, and system libraries. Instrumenting full Zephyr firmware while preserving correct execution semantics represents a substantial engineering effort beyond the scope of this work. Despite these challenges, we conducted a preliminary experiment on a Zephyr-based firmware configuration, obtaining an approximate **30%** increase in binary size after instrumentation. Overall, these experiments demonstrate that RemOTA supports *real-world code compatibility* and can effectively detect vulnerabilities in widely used firmware frameworks, while quantitative performance evaluation is carried out using the BEEBS benchmark suite.

7.4. RQ2: System usability & Performance

A key aspect of RemOTA is its performance, particularly its impact on the firmware's execution time. UART communication often dominates end-to-end latency in embedded deployments. For this reason, RemOTA is explicitly designed as an asynchronous attestation system, in which firmware execution proceeds independently of both communication and remote verification. In particular, the firmware never blocks on sending trace data nor on receiving verification results, thereby preserving real-time execution constraints even when communication is slow. Consequently, we separate two orthogonal costs in our evaluation. First, we measure on-device execution overhead, which captures the impact of instrumentation on firmware performance and is directly comparable to prior on-device protection mechanisms such as BLAST and IPEA. Second, we report the end-to-end time-to-detection, which includes communication latency and verifier processing, and reflects how quickly a temporal memory error can be detected after it occurs. This separation is necessary because communication latency is deployment-dependent and not intrinsic to RemOTA's design. While UART represents a worst-case, low-bandwidth channel commonly used during prototyping, the same instrumentation and protocol can operate over faster interfaces such as SPI, CAN, or Ethernet, significantly reducing detection latency without affecting firmware execution. To ensure a fair comparison with the state of the art in embedded protection systems (Sun et al., 2020; Yadav and Ganapathy, 2023a; Shi et al., 2024), we first focus on on-device performance, excluding the overhead introduced by the communication channel. However, to evaluate the responsiveness of our system, we also measure the Time-to-Detection the time elapsed between the occurrence of an attack and its detection by the verifier.

Table 5
Comparison of binary size increase due to instrumentation with RemOTA.

Firmware	Baseline size (byte)	Instrumented size (byte)	Size increment (%)
aha_mont64	23 760	26 028	9.55
crc32	23 184	24 148	4.16
edn	25 536	27 860	9.10
fir	25 260	26 552	5.11
libhuffbench	25 080	27 000	7.66
matmult-int	30 248	31 212	3.19
miniver	27 472	28 444	3.54
nbody	25 832	27 276	5.59
nettle-aes	35 516	40 560	14.20
nettle-sha256	26 896	28 844	7.24
primecount	22 344	24 084	7.79
sglib-combined	36 948	37 904	2.59
st	25 776	28 132	9.14
tarfind	23 132	25 712	11.15
ud	25 664	26 620	3.73
Average	**26 843**	**28 691**	**6.92**

Execution overhead

As shown in Table 4, RemOTA introduces only a geometric mean overhead of approximately 4.47% on firmware execution time. To avoid over-interpreting a single aggregate metric, we report both arithmetic and geometric means and provide a full per-benchmark breakdown. We use the geometric mean to summarize normalized overheads across benchmarks with heterogeneous runtimes, enabling comparison with prior embedded attestation systems such as OAT, C-FLAT, and BLAST. However, geometric means should not be interpreted in isolation, especially in the presence of benchmarks with double-digit overheads; for this reason, per-benchmark results are explicitly reported. Apparent speedups observed in a small number of cases are due to measurement noise and microarchitectural effects, and benchmarks with 0% overhead are excluded from the geometric mean. While a direct comparison with other remote attestation systems such as BLAST and IPEA is not possible due to differences in focus and deployment models, some qualitative observations can still be made. BLAST targets spatial memory errors (e.g., overflows) rather than temporal ones like UAF, and IPEA is a sanitizer primarily designed for testing environments rather than production deployment. Nonetheless, RemOTA significantly outperforms both in terms of efficiency: BLAST (Yadav and Ganapathy, 2023a) reports a geometric mean overhead of 54%, and IPEA (Shi et al., 2024) incurs up to 86%. It is also important to note that while IPEA provides extensive memory error detection capabilities, this comes at the cost of substantial runtime overhead and is feasible only when debugging features are enabled—an assumption incompatible with production firmware. Moreover, IPEA lacks protection guaranties under realistic attack conditions, as it does not enforce any runtime defense once deployed.

Firmware size overhead

To thoroughly assess the impact of RemOTA on the binary size, we measured and recorded the size of each firmware binary both before and after the application of the instrumentation. The results of this evaluation, presented in Table 5 and Table 7, are categorized into different segments for clarity. Notably, with RemOTA instrumentation applied, the `.bss` segment experiences an increase of 256 bytes, which aligns with the first-level non-secure (NS) cache size. In contrast, no additional space is required in the `.data` segment. Most of the instrumentation footprint resides within the `.text` segment, as additional

code is required at each checkpoint to store the checkpoint identifier in the cache.

On average, we introduced 15.6 static instrumentation points per firmware on average, covering both checkpoint locations and memory allocation functions. The cumulative space overhead in the analyzed data set shows an increase of approximately 6.92% compared to the original binary size. Furthermore, for the S-world overhead, a consistent additional space consumption is observed, with 832 bytes added to the `.text` segment and an additional 2 KB allocated for the second-level secure cache. These findings demonstrate the space implications of REMOTA and highlight the trade-off between enhanced tracking capabilities and binary size expansion. Such results are in line with the other state-of-the-art systems such as OAT, BLAST, and IPEA.

Verifier performance

We conducted a series of experiments to validate that the verifier does not introduce performance bottlenecks due to virtual execution and that it can efficiently keep up with remote firmware running on the embedded device. Given the inherent latency of slower communication channels such as UART, we opted to dump the entire trace generated by the instrumented firmware and process it in bulk. This strategic decision allowed us to obtain precise measurements of both parsing and virtual execution speeds for individual instructions within the verifier. Our results indicate that each instruction requires an average execution time of 76 microseconds, which demonstrates the efficiency of the verifier in handling the workload. This selective execution strategy ensures that the reported 76 microseconds per instruction is a robust metric, further underscoring the verifier's ability to manage real-time workloads without compromising performance. By focusing only on checkpoint-related instructions, the verifier efficiently optimizes resource utilization while maintaining accurate monitoring capabilities.

End-to-end system performance scalability

In addition to performance evaluation, we conducted an extensive 24-hour stress test to assess memory allocation behavior under continuous load. For this experiment, we selected the nettle-sha256 firmware from our dataset and subjected it to a repetitive execution cycle until the predefined time limit was reached. The test began with an initial memory allocation of 8 MB, which incrementally expanded to approximately 19 MB to accommodate the core data structures of the firmware. Once this initial expansion phase was complete, memory usage stabilized and remained consistent throughout the remainder of the test period. This outcome is encouraging, as it indicates that the verifier effectively manages memory without excessive or unexpected growth. Furthermore, given that the verifier is intended to operate in a server environment, the observed memory consumption falls well within acceptable limits. These results confirm that the verifier is both efficient and scalable, ensuring reliable operation even under prolonged and intensive workloads.

Time-to-detection benchmark

The goal of this experiment is to evaluate the time elapsed between the moment an attack is executed on the device and its detection by the remote verifier. Table 6 presents the time-to-detection measurements for two firmware configurations: one with a minimal cache (two elements) and another with a fully populated cache. The reported times encompass the entire operational pipeline firmware execution, UART transmission, and remote verifier evaluation. When the cache contains only a single element, transmission rate fluctuations are minimal, and the average time-to-detection is approximately 0.475 s, involving one Secure and one NS cache flush. In the worst-case scenario, where the cache holds 1022 elements, the NS cache flushes eight times while the Secure cache flushes once. In this case, transmission speed becomes a critical factor lower data rates significantly increase the overall detection time. Even when isolating firmware execution, UART transmission dominates the total latency. Verifying a fully populated cache takes approximately 465 ms. The communication overhead is inherently dependent on the firmware characteristics, including the number and frequency of triggered instrumentation points, as well as the application's own communication patterns. Therefore, its impact must be evaluated on a per-deployment basis.

Aliasing and linked structures

Although REMOTA does not aim to provide a fully comprehensive alias analysis, we evaluate its behavior on common aliasing patterns involving heap-allocated structures. In particular, we tested scenarios where a pointer alias references a freed node in a linked structure. REMOTA successfully detects the resulting use-after-free, as detection relies on dynamic virtual execution rather than solely on static alias resolution. This confirms that the system handles practical aliasing patterns commonly found in embedded firmware, despite not covering all possible aliasing cases.

7.5. RQ3: Security analysis

Our threat model considers the possibility that attackers may discover and exploit previously unknown vulnerabilities in the embedded programs running within the Normal World. To successfully bypass REMOTA, an attacker would need to perform one of the following actions: (a) disable the instrumentation or trampolines, (b) manipulate the instrumentation parameters, (c) manipulate the execution trace, including replaying attacks or (d) cache attacks. In the following sections, we analyze each potential attack vector in detail and explain how REMOTA mitigates these threats.

Disabling the instrumentation. For an attacker to disable or circumvent the instrumentation mechanisms, they would need to modify the embedded code. However, this is infeasible under our security assumptions, as code integrity measures, such as secure boot and memory protection units (MPUs), are in place to prevent unauthorized modifications. Additionally, to evade detection, an attacker would have to manipulate the control flow of the application before the instrumentation executes. However, since the instrumented code is triggered early and its execution is recorded in the trace, such attempts would be logged and subsequently detected during the verification process. This ensures that any deviation from the expected execution flow is flagged as suspicious.

Misusing instrumentation parameters. An attacker attempting to compromise the parameters passed to the trampoline functions would need to manipulate checkpoint identifiers, such as node IDs. Our design effectively mitigates this risk by embedding statically inferred values, such as node IDs, as immediate values in the read-only `.text` segment of the program during compilation. This placement ensures immutability and prevents runtime tampering. Additionally, dynamic run-time attributes, such as allocation base addresses and sizes, are securely recorded within the Checkpoint events immediately after buffer creation. By storing these critical values in the S world before any potential overflow or attack can occur, REMOTA ensures the integrity of memory tracking. To protect the parameters of the instrumentation points, we can adopt a similar protection mechanism inspired by SafeStack (SafeStack, 0000). A randomly allocated region securely stores sensitive variables, such as checkpoint IDs, providing an additional layer of protection against memory corruption attacks.

Altering or replaying execution traces. To guarantee the integrity of the execution trace and protect against tampering or replay attacks, our system employs a cryptographically secure attestation mechanism. The attestation blob, which contains essential runtime information, is signed using a hardware-provisioned private key securely stored thanks to One-Time Programmable (OTP) offered by STM32L5 and Read out Protection (RDP). Verifiers can validate the attestation data by checking the authenticity of the signature, ensuring that the trace

Table 6
Execution times for test firmware under minimal and maximal cache usage, showing the impact of UART transmission and the end-to-end time taken in the fullchain (firmware execution + UART transmission + verify).

Execution time	Cache state	CP	Tx size	Cache flush		Baud rate			
				NS	S	**9600**	**19 200**	**115 200**	**1 152 000**
End-to-end	Almost empty (best)	2	8B	1	1	0.47 s	0.53 s	0.47 s	0.43 s
	Full (worst)	1022	2000B	8	1	2.62 s	1.66 s	0.68 s	0.53 s
Firmware only	Almost empty (best)	2	8B	1	1	14 ms	8 ms	2 ms	1 ms
	Full (worst)	1022	2000B	8	1	2234 ms	1165 ms	275 ms	114 ms
Remote Verifier evaluation time						**386** ms	**495** ms	**405** ms	**416** ms

remains unaltered and trustworthy. Additionally, RemOTA incorporates nonce-based verification to prevent replay attacks. Each attestation request includes a unique cryptographic nonce generated by the verifier, and the attestation blob must contain the correct nonce value to be considered valid. Any attempt to replay an old attestation will fail due to a nonce mismatch, providing a robust safeguard against replay-based attacks.

Cache attacks. The RemOTA architecture incorporates a two-level caching mechanism to strike a balance between performance and security. It deploys one cache in the S, which benefits from hardware-enforced isolation, and another in the NS, which is considered untrusted and may be subject to attacker manipulation. Protecting sensitive data in NS presents a well-known challenge. Existing solutions often rely on validating memory integrity by checking and masking values immediately after store operations. However, these techniques require instrumentation of all store instructions, which is impractical in our context due to the significant execution overhead particularly in resource-constrained embedded systems. To address this, RemOTA implements a more lightweight and practical cache design. Cache entries in the N world are allocated at random memory locations, making it significantly harder for an attacker to predict or target sensitive data. Once the cache reaches a predefined threshold, its contents are flushed into the Secure L2 cache located in S. Additionally, data stored in the NS cache does not persist for extended periods; the cache is periodically flushed to reduce the time window during which data could be exposed or tampered with. This approach minimizes the opportunity for attackers to interfere with the caching layer, while avoiding the high overhead associated with fully instrumented store-level protection. It provides a practical and efficient mechanism to safeguard transient sensitive data during execution, without compromising system performance.

8. Discussion

In this section, we provide a detailed discussion of the current limitations of RemOTA and outline potential challenges and areas for improvement in future work.

Interrupt handling & Multi-threads

One of the significant limitations of the prototype RemOTA is its lack of support for interrupt handling functions. Currently, the instrumentation process is limited to functions that are directly invoked by the firmware entry point, thereby excluding interrupt-driven operations from analysis. This constraint means that any functionality triggered asynchronously by hardware interrupts remains outside the verification scope of RemOTA. To address this limitation, interrupt handling functions could be integrated into the initial set of functions designated for instrumentation. By doing so, the analysis would expand to cover all possible interactions within these functions, ensuring a more comprehensive verification process. In particular, the verifier itself would not require any changes as interrupt handling functions would be treated similarly to other functions. Implementing interrupt support is primarily an engineering effort that can be addressed in future iterations without necessitating significant architectural modifications to the system design. Incorporating this feature would enhance the overall robustness and applicability of RemOTA in embedded environments of the real world.

Pointer aliasing within structures

Another important limitation of the current implementation RemOTA is the lack of support for pointer aliasing within structures. The RemOTA verifier employs a lightweight chunk allocation mechanism that does not track the internal content of allocated memory chunks. Instead, the RemOTA allocation table consists solely of metadata and does not contain any actual data. As a result, when a pointer within a chunk is accessed, it cannot yield a meaningful value, since the chunk itself does not maintain internal data representation. Supporting pointer aliasing within structures would require implementing a more sophisticated memory tracking approach, such as a one-to-one shadow memory model. However, such an approach would introduce significant overhead and complexity, which is not feasible in the resource-constrained environments in which RemOTA operates. The current lightweight approach is intentionally designed to maintain efficiency and minimize performance impact, making it well-suited for real-time embedded systems.

Future directions

Despite these limitations, RemOTA presents a strong foundation for secure firmware verification. Future efforts could explore optimizing the instrumentation process to seamlessly incorporate interrupt handling functions while maintaining low overhead. Additionally, investigating alternative memory tracking techniques that balance accuracy and efficiency could provide a feasible solution for handling pointer aliasing within structures. In conclusion, while RemOTA offers a practical and lightweight solution for the verification of embedded systems, addressing these limitations will be crucial to broadening its applicability and enhancing its effectiveness in more complex and dynamic environments. Future improvements will require a combination of engineering refinements and potential architectural improvements to achieve a more comprehensive and robust verification framework.

Applicability

Based on our evaluation, RemOTA is best suited for firmware that (i) makes non-trivial use of dynamic memory allocation, (ii) must strictly limit on-device overhead and therefore cannot rely on pervasive instrumentation, and (iii) runs on platforms equipped with a hardware-enforced trust anchor such as TrustZone. In particular, firmware exhibiting complex control flow and conditional pointer updates consistently benefits from RemOTA's checkpoint-based design, which enables precise detection of temporal memory errors while maintaining low runtime cost. In contrast, firmware that relies exclusively

Table 7
Comparison of binary size increase due to instrumentation with RemOTA. The table includes the baseline size, the additional space required by RemOTA instrumentation and the number of static instrumentation points for each firmware.

Firmware	Static instrumentation				Normal				RemOTA				Size
	Alloc	CP	Use	Paths	.text	.data	.bss	Total	.text	.data	.bss	Total	(%)
aha_mont64	3	33	235	97	21 424	104	2232	23 760	23 436	104	2488	26 028	9.55
st	2	22	115	52	23 424	120	2232	25 776	25 524	120	2488	28 132	9.14
edn	2	16	91	30	22 416	104	3016	25 536	24 476	104	3280	27 860	9.10
fir	2	8	16	13	22 956	104	2200	25 260	23 984	104	2464	26 552	5.11
nettle-aes	3	40	739	77	32 412	392	2712	35 516	37 192	392	2976	40 560	14.20
libhuffbench	2	20	28	41	22 272	104	2704	25 080	23 936	104	2960	27 000	7.66
ud	2	2	19	14	21 592	104	3968	25 664	22 292	104	4224	26 620	3.73
nbody	2	8	19	14	23 192	440	2200	25 832	24 372	440	2464	27 276	5.59
minver	1	4	3	5	24 984	176	2312	27 472	25 692	176	2576	28 444	3.54
matmult-inc	1	4	3	5	19 944	104	10 200	30 248	20 644	104	10 464	31 212	3.19
tarfind	1	22	74	40	20 828	104	2200	23 132	23 144	104	2464	25 712	11.15
nettle-sha256	1	10	40	14	24 536	160	2200	26 896	26 220	160	2464	28 844	7.24
sglib-combined	0	2	0	1	25 948	104	10 896	36 948	26 648	104	11 152	37 904	2.59
primecount	3	14	47	27	20 040	104	2200	22 344	21 516	104	2464	24 084	7.79
crc32	1	4	3	5	20 872	104	2208	23 184	21 580	104	2464	24 148	4.16
Average	1.75	13.94	90.38	28.06	23 123	155	3565	26 843	24 710	155	3826	28 691	6.92

on static allocation, executes only briefly, or can tolerate substantial instrumentation overhead may not require RemOTA's remote attestation model and may be more effectively served by static analysis or testing oriented sanitizers.

CRediT authorship contribution statement

Matteo Zoia: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Mirco Picca:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Davide Rusconi:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Andrea Monzani:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Flavio Toffalini:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Danilo Bruschi:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Andrea Lanzi:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Andrea Lanzi reports administrative support and travel were provided by SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Andrea Lanzi reports a relationship with SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU that includes: travel reimbursement. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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. The authors would like to thank Cristian Salvi for his contributions to the testing and experimental validation of the proposed system.

Data availability

Dataset and source code are available, link in the paper references.

References

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

Almakhdhub, N.S., Clements, A.A., Bagchi, S., Payer, M., 2020. *mu*RAI: Securing embedded systems with return address integrity. In: Network and Distributed Systems Security (NDSS) Symposium.

Andersen, L.O., 1994. Program Analysis and Specialization for the C Programming Language (Ph.D. thesis). Citeseer.

Arm Ltd, 2016. System design with ARMv8-M. https://developer.arm.com/docs/100767/0100/system-design-for-armv8m.

Austin, T.M., Breach, S.E., Sohi, G.S., 1994. Efficient detection of all pointer and array access errors. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. PLDI '94, Association for Computing Machinery, New York, NY, USA, pp. 290–301. http://dx.doi.org/10.1145/178243.178446.

Ba, J., Duck, G.J., Roychoudhury, A., 2023. Efficient greybox fuzzing to detect memory errors. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3551349.3561161.

Berger, E.D., Zorn, B.G., 2006. DieHard: probabilistic memory safety for unsafe languages. SIGPLAN Not. 41 (6), 158–168. http://dx.doi.org/10.1145/1133255.1134000.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In: NDSS. pp. 1–15.

Clements, A.A., Almakhdhub, N.S., Bagchi, S., Payer, M., 2018. {ACES}: Automatic compartments for embedded systems. In: 27th USENIX Security Symposium. USENIX Security 18, pp. 65–82.

Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., 2014. A {Large-scale} analysis of the security of embedded firmwares. In: 23rd USENIX Security Symposium. USENIX Security 14, pp. 95–110.

Ding, R., Qian, C., Song, C., Harris, B., Kim, T., Lee, W., 2017. Efficient protection of path-sensitive control security. In: 26th USENIX Security Symposium. USENIX Security 17, USENIX Association, Vancouver, BC, pp. 131–148, URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding.

Du, Y., Guo, Y., Zhang, Y., Yang, J., 2024. RTT-UAF: Reuse time tracking for use-after-free detection. In: Proceedings of the 38th ACM International Conference on Supercomputing. ICS '24, Association for Computing Machinery, New York, NY, USA, pp. 376–387. http://dx.doi.org/10.1145/3650200.3656606.

Du, Y., Shen, Z., Dharsee, K., Zhou, J., Walls, R.J., Criswell, J., 2022. Holistic control-flow protection on real-time embedded systems with kage. In: 31st USENIX Security Symposium. USENIX Security 22, USENIX Association, Boston, MA, pp. 2281–2298, URL: https://www.usenix.org/conference/usenixsecurity22/presentation/du.

Duck, G.J., Yap, R.H.C., Cavallaro, L., 2017. Stack bounds protection with low fat pointers. In: Network and Distributed System Security Symposium. URL: https://api.semanticscholar.org/CorpusID:4960605.

Gorter, F., Koning, K., Bos, H., Giuffrida, C., 2022. DangZero: Efficient use-after-free detection via direct page table access. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. CCS '22, Association for Computing Machinery, New York, NY, USA, pp. 1307–1322. http://dx.doi.org/10.1145/3548606.3560625.

Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., van der Kouwe, E., 2016. TypeSan: Practical type confusion detection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, Association for Computing Machinery, New York, NY, USA, pp. 517–528. http://dx.doi.org/10.1145/2976749.2978405.

He, L., Hu, H., Su, P., Cai, Y., Liang, Z., 2022. FreeWill: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In: 31st USENIX Security Symposium. USENIX Security 22, USENIX Association, Boston, MA, pp. 2497–2512, URL: https://www.usenix.org/conference/usenixsecurity22/presentation/he-liang.

Hind, M., Pioli, A., 2000. Which pointer analysis should I use? SIGSOFT Softw. Eng. Notes 25 (5), 113–123. http://dx.doi.org/10.1145/347636.348916.

Jeon, Y., Han, W., Burow, N., Payer, M., 2020. FuZZan: Efficient sanitizer metadata design for fuzzing. In: 2020 USENIX Annual Technical Conference. USENIX ATC 20, USENIX Association, pp. 249–263, URL: https://www.usenix.org/conference/atc20/presentation/jeon.

Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., Lee, W., 2015. Preventing use-after-free with dangling pointers nullification. In: NDSS.

Li, Y., Tan, W., Lv, Z., Yang, S., Payer, M., Liu, Y., Zhang, C., 2022. PACMem: Enforcing spatial and temporal memory safety via ARM pointer authentication. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. CCS '22, Association for Computing Machinery, New York, NY, USA, pp. 1901–1915. http://dx.doi.org/10.1145/3548606.3560598.

Microelectronics, S., 2020. Reference manual. https://www.st.com/en/microcontrollers-microprocessors/stm32l5-series.html.

Moghadam, V.E., Meloni, M., Prinetto, P., 2021. Control-flow integrity for real-time operating systems: Open issues and challenges. In: 2021 IEEE East-West Design & Test Symposium. EWDTS, pp. 1–6. http://dx.doi.org/10.1109/EWDTS52692.2021.9581003.

Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S., 2009a. SoftBound: highly compatible and complete spatial memory safety for c. SIGPLAN Not. 44 (6), 245–258. http://dx.doi.org/10.1145/1543135.1542504.

Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S., 2009b. SoftBound: highly compatible and complete spatial memory safety for c. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '09, Association for Computing Machinery, New York, NY, USA, pp. 245–258. http://dx.doi.org/10.1145/1542476.1542504.

Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S., 2010. CETS: compiler enforced temporal safety for C. In: Proceedings of the 2010 International Symposium on Memory Management. ISMM '10, Association for Computing Machinery, New York, NY, USA, pp. 31–40. http://dx.doi.org/10.1145/1806651.1806657.

Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W., 2005. CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. 27 (3), 477–526. http://dx.doi.org/10.1145/1065887.1065892.

Ning, Z., Zhang, F., 2017. Ninja: Towards transparent tracing and debugging on ARM. In: 26th USENIX Security Symposium. USENIX Security 17, USENIX Association, Vancouver, BC, pp. 33–49, URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning.

NIST, N.C.f.A.S., 2017. Juliet C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112.

Pallister, J., Hollis, S., Bennett, J., 2013. BEEBS: Open benchmarks for energy measurements on embedded platforms. arXiv:1308.5174, https://arxiv.org/abs/1308.5174.

Rusconi, D., Zoia, M., Buccioli, L., Pierazzi, F., Bruschi, D., Cavallaro, L., Toffalini, F., Lanzi, A., 2024. EmbedWatch: Fat pointer solution for detecting spatial memory errors in embedded systems. In: Proceedings of the Sixth Workshop on CPS&IoT Security and Privacy. CPSIoTSec '24, Association for Computing Machinery, New York, NY, USA, pp. 55–67. http://dx.doi.org/10.1145/3690134.3694815.

SafeStack, SafeStack. https://clang.llvm.org/docs/SafeStack.html.

Salehi, M., Hughes, D., Crispo, B., 2020. uSBS: Static binary sanitization of bare-metal embedded devices for fault observability. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses. RAID 2020, USENIX Association, San Sebastian, pp. 381–395, URL: https://www.usenix.org/conference/raid2020/presentation/salehi.

Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. AddressSanitizer: A fast address sanity checker. In: 2012 USENIX Annual Technical Conference. USENIX ATC 12, USENIX Association, Boston, MA, pp. 309–318, URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

Shi, J., Li, W., Wang, W., Guan, L., 2024. Facilitating non-intrusive in-vivo firmware testing with stateless instrumentation. In: 31st Network and Distributed System Security Symposium. NDSS.

Slamaris, D., Embedded systems security and trustzone. https://embeddedsecurity.io/sec-tz-basics.

Steensgaard, B., 1996. Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 32–41.

Sui, Y., Xue, J., 2016. SVF: interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 265–266.

Sun, Z., Feng, B., Lu, L., Jha, S., 2020. OAT: Attesting operation integrity of embedded devices. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1433–1449.

Szekeres, L., Payer, M., Wei, T., Song, D., 2013. Sok: Eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. SP '13, IEEE Computer Society, USA, pp. 48–62. http://dx.doi.org/10.1109/SP.2013.13.

Wang, Y., Lemieux-Mack, C., Chantem, T., Baruah, S., Zhang, N., Ward, B.C., 2024. Partial context-sensitive pointer integrity for real-time embedded systems. In: 2024 IEEE Real-Time Systems Symposium. RTSS, pp. 415–426. http://dx.doi.org/10.1109/RTSS62706.2024.00042.

Wickman, B., Hu, H., Yun, I., Jang, D., Lim, J., Kashyap, S., Kim, T., 2021. Preventing use-after-free attacks with fast forward allocation. In: 30th USENIX Security Symposium. USENIX Security 21, USENIX Association, pp. 2453–2470, URL: https://www.usenix.org/conference/usenixsecurity21/presentation/wickman.

Yadav, N., Ganapathy, V., 2023a. Whole-program control-flow path attestation. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 2680–2694.

Yadav, N., Ganapathy, V., 2023b. Whole-program control-flow path attestation. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. CCS '23, Association for Computing Machinery, New York, NY, USA, pp. 2680–2694. http://dx.doi.org/10.1145/3576915.3616687.

Yong, S.H., Horwitz, S., 2003. Protecting C programs from attacks via invalid pointer dereferences. SIGSOFT Softw. Eng. Notes 28 (5), 307–316. http://dx.doi.org/10.1145/949952.940113.

Zhang, H., Kim, J., Yuan, C., Qian, Z., Kim, T., 2025. Statically discover cross-entry use-after-free vulnerabilities in the linux kernel. In: Network and Distributed System Security (NDSS) Symposium.

Zhang, T., Lee, D., Jung, C., 2019. BOGO: Buy spatial memory safety, get temporal memory safety (almost) free. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '19, Association for Computing Machinery, New York, NY, USA, pp. 631–644. http://dx.doi.org/10.1145/3297858.3304017.

Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., Walls, R.J., 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In: 29th USENIX Security Symposium. USENIX Security 20, pp. 1219–1236.

Zoia, M., et al., 2026. Source code of REMOTA. https://zenodo.org/records/15190636.