Received: 22 September 2022 | Revised: 31 December 2022 | Accepted: 24 February 2023

IET Information Security

DOI: 10.1049/ise2.12113

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH

Hardware-assisted remote attestation design for critical embedded systems

Munir Geden | **Kasper Rasmussen**

Department of Computer Science, University of Oxford, Oxford, UK

Correspondence

Munir Geden and Kasper Rasmussen.
Email: munir.geden@cs.ox.ac.uk and kasper.rasmussen@cs.ox.ac.uk

Abstract

Remote attestation, as a challenge-response protocol, enables a trusted entity, called *verifier*, to ask a potentially infected device, called *prover*, to provide integrity assurance about its internal state. Remote attestation is becoming increasingly vital for embedded systems that serve in many critical domains, as part of health, military, transportation and industry services, but still lack the most security features available to high-end systems. In most attestation techniques, the prover provides a cryptographic checksum of its static memory contents, that is, code segments, to the verifier when requested to demonstrate that the device is loaded with the right software. However, those measurements are subject to two limitations. First, they cannot guarantee that the prover has always had legitimate software in the memory prior to attestation. This is because occasional measurements, triggered by the verifier, still leave the device vulnerable to the compromise between two attestation windows as a time-of-check-to-time-of-use (TOCTOU) problem. Second, including dynamic memory regions in the checksum calculation is not helpful in practice, since the verifier typically does not know what those regions should contain or which checksums should be accepted as valid. Hence, many attack scenarios residing in those dynamic regions (e.g. stack) would also go unnoticed. To reveal attack scenarios exploiting the memory regions and time windows left unattested, we propose an attestation scheme that can continuously monitor both static and dynamic memory regions with better spatial and temporal attestation coverage. Our monitoring mechanism is designed to be performed in real time using a novel hardware security module (HSM) connected to the prover's system bus. The proposed HSM monitors not only the integrity of the code on the prover but also its execution by checking the compliance of the bits seen on the bus according to a runtime integrity model (RIM) of the prover's software. Therefore, our attestation scheme is capable of reporting scenarios that violate both the (static) code and (dynamic) runtime integrity since the deployment time.

KEYWORDS

embedded systems, protocols, security

1 | INTRODUCTION

Remote attestation aims to address these risks by providing reports on the integrity of a device to a remote entity. A remote attestation scheme generally consists of two parties. *Prover*, as a potentially infected device, has to assure a remote party called *verifier* that the device is in a benign state. In a typical attestation scheme, the verifier makes a request to the prover with a challenge. Then, the prover performs some measurements on its memory and returns it as a signed response. Upon receiving the response, if satisfied with its freshness, integrity and authenticity, the verifier can then decide whether the prover is in a legitimate state using the measurement returned.

In conventional attestation schemes, when the verifier makes a request, the prover calculates a cryptographic checksum of its static memory contents (i.e., code segments) and returns it to the verifier as a proof. However, there are two limitations of such an approach. The first one is that occasional measurements triggered by the verifier cannot guarantee that the prover has always been in the proven state. Due to the lack of continuous monitoring, an attack scenario that starts and finishes between two measurement windows would not be caught as long as the attacker leaves the attested memory regions in an acceptable state, that is, time-of-check-to-time-of-use (TOCTOU) attacks. Unfortunately, attempts to shorten these time gaps through more frequent attestation requests by the verifier would have a significant performance impact on the availability aspect of the device because the prover must spend most of its execution time on checksum calculations [1].

The second limitation with the checksum-based approach is the applicability to dynamic memory regions (e.g., stack), where many attacks, such as return-oriented programming (ROP) scenarios, can be accommodated. Unlike static code regions, a single checksum of dynamic regions at a certain time would not deliver practical value to the verifier for mainly two reasons: The first one is that the verifier cannot simply reason about such a measurement unless the verifier has access to the same (external) program input with the same hardware and software settings. This is because each execution or corresponding state at a particular time would be specific to the external data provided, such as environment and user input, which makes the problem undecidable from the verifier's perspective. The second reason is that even if all external data are excluded from checksums, discovering all acceptable checksums would still be impractical for many programs due to the combinatorial explosion of internal variable values. Therefore, the checksum returned would be inconclusive.

Many attestation schemes fail to address those together and ignore attacks that can exploit time gaps or memory regions left unattested. There have been attempts to address some of those limitations. A recent work RATA [1] elegantly addresses the TOCTOU attacks on static regions by including their last modification time in the attestation measurement. Likewise, this approach is not applicable to dynamic memory regions since the verifier does not know what these regions should contain or when they should be written. On the other hand, an increasing number of runtime attestation work [2–4] suggests providing a cumulative hash of path traces to inform the verifier about the states observed in dynamic regions, so control flow attacks that corrupt them can be revealed. Because the prover returns only a single hash representing the whole program execution, those schemes require the verifier to discover all possible control-flow traces and corresponding hashes in advance from the program's control-flow graph (CFG). However, this requirement overlooks potential challenges on the verifier side due to the same reasons mentioned above, which are the rapid explosion of path search space for many programs, and the undecidability of the verification problem without program input in case of attacks complying with the CFG (i.e. control-flow bending). We note that program input is typically determined by external agents, such as the environment and users. Assuming that the verifier provides the program input [3] that can eliminate the need for having a prover device as the same computation could have been performed on the verifier as the trusted party.

To address these drawbacks in a more practical setting, this paper proposes an attestation scheme that monitors the prover with the help of a hardware security module (HSM) connected to its system bus. The HSM is responsible for measuring both code and its execution according to a runtime integrity model (RIM) provided by the verifier to the HSM. Thanks to its continuous monitoring, our scheme promises to catch any TOCTOU cases that can temporarily alter the device software even for a short time. Furthermore, it substitutes trace-based checks on the verifier side with model-based checks in real time for a more efficient approach. Therefore, our scheme does not require the verifier to generate traces in advance or to have access to the same environment/user input. Also, thanks to the use of an invasive off-chip hardware module, it offers an attestation solution that can fit better into legacy systems.

We previously suggested a similar setting in the conference version of this work [5], that is, a hardware module (HSM) connected to the prover's system bus to monitor and attest software runtime in real time according to a static runtime model (RIM). However, the conference version was quite restrictive about the software subject to the attestation. For instance, it was not allowing any use of recursive functions and was accepting only software instances whose static model could be extracted with high precision. Also, the conference version was more demanding in terms of HSM resources, such as memory requirements. In contrast, this paper proposes a more practical solution with completely changed RIM and HSM designs that can fit into a more realistic setting, accepting a wider range of software instances. Regarding the attestation scope, this version attests not only program runtime but also to the program code, unlike the previous work obtaining code integrity by assumption. In terms of program models (RIM), this paper replaces the previous branch-centric model with a call-centric approach that focuses on only control flow events that matter most. In addition, the HSM, which was previously monitoring program execution mainly through instruction addresses, is redesigned to iterate by decoding specific instructions available on the data bus as detailed in Section 4. The new design also replaces previous fine-grained data flow checks with more coarse-grained policies to avoid the inefficiency of monitoring the program execution at small block granularity. Lastly, regarding the dynamic features, this paper can follow up executions whose stack can go unbounded, (e.g., recursive functions), thanks to the elimination of shadow stack use.

In line with this vision, this paper makes the following contributions:

- Attestation of both programme code and execution with strong spatial and temporal coverage.
- Approximation of programme runtime via a more lightweight static model (i.e., RIM) that can be hosted with resource-constrained devices.

- An off-chip hardware module with a runtime monitoring logic that can address strong adversary assumptions and legacy issues of critical embedded systems.

The rest of the paper is organised as follows: Section 2 defines the problem scope and the assumptions about the system and the adversary. Section 3 describes the runtime integrity model of the software subject to attestation and Section 4 explains how the HSM uses this model to check the correctness of runtime in real time. Section 5 explains the details of protocol reporting about the prover's state of the prover. Finally, Sections 6 and 7 analyse the security and performance aspects of the proposed scheme. Sections 8 and 9 review related work and provide a discussion on possible extensions and the energy impact.

2 | PROBLEM SETTING

In a conventional attestation protocol where the verifier initiates the process with a nonce and the prover responds with a checksum measurement of its static memory regions, we can count two main limitations: The first one is the lack of strong temporal coverage. A memory measurement triggered by the verifier's request can best prove that the static code regions are in a good state by the time the request is received. However, such occasional measurements do not provide any information about the time the verifier is idle as shown in Figure 1. Therefore, an attacker can exploit the gap between two attestation windows and can execute corrupted or malicious codes without being caught as a time-of-check-to-time-of-use (TOCTOU) problem. Although a valid checksum implies that the prover has a genuine code at the time of the check, we cannot guarantee that the code (used) has always been in that state unless all modifications to the code regions are recorded and reported [1]. In this paper, we refer to these scenarios as *code attacks*. Another drawback of static attestation is the weak spatial coverage of the memory. Dynamic memory regions are not typically included in checksums since the verifier cannot easily reason about their contents, often containing user and environment data also unknown to the verifier. Hence, the prover can be compromised by a simple *code-injection* or a more sophisticated *code-reuse* attack that touches only those addresses (e.g., stack). For instance, return-oriented programming (ROP) techniques crafting the stack with the required return address and data can still achieve arbitrary code execution on the prover without altering attested code blocks.

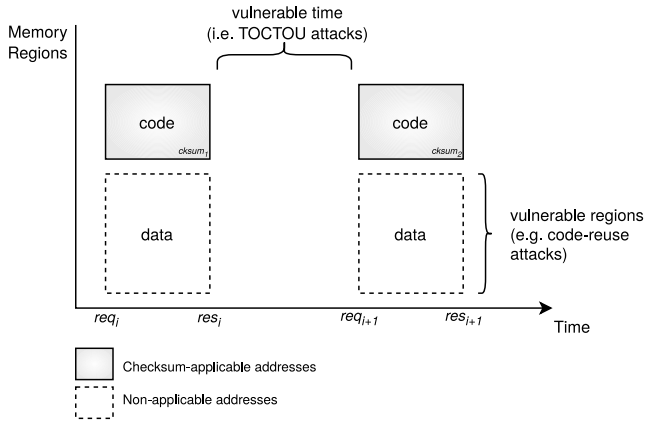

FIGURE 1 Limitations of conventional checksum-based static attestation schemes.

2.1 | Code attacks

In the absence of continuous monitoring of code addresses [1], an attacker can temporarily compromise the prover's software by altering or replacing it with malicious code, and can switch it back to the expected state prior to following attestation request. The attacker would have different options, such as *memory copy* and *hiding* techniques to act between two attestation measurements without being noticed. The attacker can utilise free memory on the prover to keep both malicious and genuine codes simultaneously [6]. He can thus calculate a checksum from the original code when requested despite the use of malicious one prior to the checksum measurement. If there is not enough space for hosting two code instances, *data substitution* [7] techniques can keep only the record of changes and can revert them during the measurement time. Alternatively, *compression* [8] methods can provide extra space to host both code instances, whereas valid measurements are provided through on-the-fly decompression. In different settings, *proxy* attacks can also benefit from a more resourceful device to hold the copy of original contents on. Thus, the compromised prover can forward the request to the proxy node to produce a valid measurement and impersonate the prover node with a valid response.

2.2 | Code-reuse attacks

The prover can still be compromised even if it is loaded solely with the genuine programme code all the time. An attacker modifying programme data (e.g., return addresses) can maliciously reuse the original code. With a programme that provides the necessary code snippets (i.e., attack gadgets), code-reuse attacks can be Turing-complete, meaning that any (arbitrary) code can be expressed without injecting a new code or altering the existing one. For a successful *code-reuse* scenario, the attacker generally exploits control-flow transfer instructions, the destination addresses of which are given from the data segments. For typical embedded software implemented using C language, the attacker would have many options: The first one is exploiting the return addresses on the stack [9, 10], known as *return-oriented programing (ROP)* attacks. Alternatively, the attacker can take advantage of indirect jump or indirect call transfers (e.g., function pointer) if the code contains [11, 12]. These scenarios are also called *jump-(JOP)* and *call-oriented programming (COP)* attacks. In this paper, we refer to all these types as *control attacks*. Additionally, the attacker can specifically target programme

variables [13–15] without touching control transfer destinations, such as a global flag that can result in the execution of a privileged programme path. These are also called data attacks.

2.3 | System model

For our scheme, we consider two entities: the *verifier* and the *prover*. As the remote party, the verifier is trusted and can ask the prover to provide a report showing that the prover is in a good state at will. The prover is an embedded device, such as microcontroller (MCU), without cache. The device has a single-purpose monolithic software (i.e., bare-metal). It executes instructions directly on logic hardware with physical memory addresses. The software subject to the attestation can contain indirect calls (e.g., function pointer), jumps (e.g., switch statements) and recursive functions. Although our scheme is applicable to different architectures with minor changes, the detailed design in the following sections considers a load-store architecture with fixed-length instructions (i.e., RISC).

Additionally, the prover has an off-chip low-cost hardware security module (HSM) connected to its system bus, as seen in Figure 2. The HSM illustrated in Figure 3 has built-in hardware implementations of the bus monitoring logic described in Section 4. The HSM has limited memory resources, mainly hosting a static runtime integrity model (RIM) of the programme subject to attestation. This static model, described in detail in Section 3, is provided by the verifier and loaded into the HSM during deployment. HSM's memory also contains some dynamic bits that keep track of the current (executing) function and the number of calls made from each function. These bits collaborate with the static model to monitor the runtime integrity of the programme. While the information captured through the bus provides the execution data, the HSM provides a basic attestation API that reports the device's status to the verifier. The HSM keeps a key (sk) that never leaves its internal memory and is used to sign the attestation responses.

2.4 | Adversary model

Prior to the attestation, the adversary has access to the source code, binary and RIM. External resources are available to collect or record any protocol activity for later use. Only software attacks targeting memory are considered, while physical attack capabilities on both the prover device and the HSM are beyond the scope of this paper. The adversary has the ability to write an arbitrary value to an arbitrary memory address. He can modify the programme code and put it back to the original state at any time (i.e., code attacks). He can also manipulate the programme execution by corrupting control data (i.e., code-reuse attacks) on dynamic memory regions though the adversary cannot affect the HSM's internal state and the verifier.

The ultimate goal of the adversary is malicious execution on the prover without being noticed by the verifier. Acting on the prover, the adversary can try to hide attack artefacts from the HSM. If this is not possible and the HSM has already detected an attack, the adversary may attempt to prevent the genuine reports from being received by the verifier and replace them with counterfeit but acceptable ones. The adversary can arbitrarily call the HSM's API to learn about the HSM's internal state or to generate signed attestation reports for later use. The adversary can intercept and modify any messages on the network or replay the responses sent earlier.

3 | DESIGN OF THE RUNTIME INTEGRITY MODEL (RIM)

Prior to deployment, the verifier extracts a runtime integrity model (RIM). This model approximates the benign executions that the code can have. It is stored by the HSM internally and is used as a reference model to check whether the information captured at runtime through the system bus complies with the expected behaviour. This is a two-layered static programme model centred around the call graph. The main control layer models legitimate control transfers amongst programme functions, such as calls and returns. The second layer enables coarse-grained checks on memory accesses, for instance, checking whether a function is allowed to access global variables or caller frames on the stack.

To explain the RIM and to exemplify possible attacks covered in Section 6.3, we will use the example code in Figure 4. The given code assumes a vulnerable login

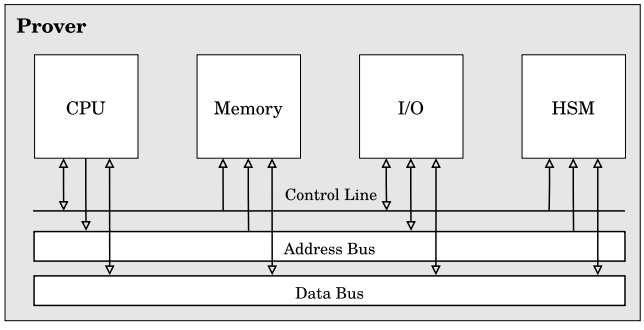

FIGURE 2 Prover's bus architecture with an off-chip hardware secure module (HSM) connected.

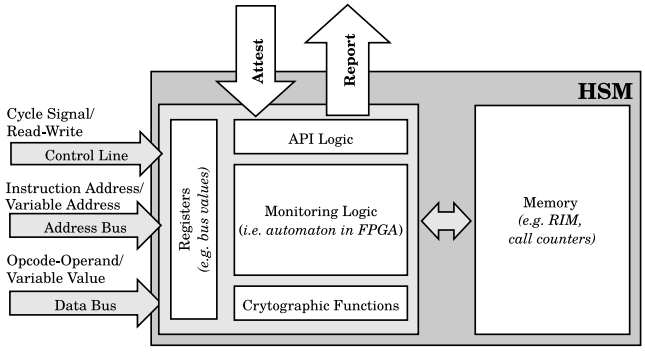

FIGURE 3 Overview of the hardware security module (HSM) illustrating its internal components and external interactions.

mechanism that can form a basis for different attack scenarios. The code illustrates that two functions login and authenticate using the global user_info variable for login status. The former function implements the core logic and creates a user session, whereas the latter function, which checks the credentials and sets the necessary info, contains a bug that provides an arbitrary memory write capability to the attacker. Despite the details omitted, the authenticate function has an indirect jump (e.g., switch statement). A corresponding RIM of this code, which is elaborated in the following sections, can be found in Figure 5.

3.1 | Static model

The RIM has different nodes and edge components to guide the HSM on what action is required for each instruction. As seen in Figure 5, nodes illustrated with squares correspond to *function blocks*, while the solid directed edges represent *control transfers* between them. These constitute the main control layer of the model. The secondary data layer is depicted by circle-shaped nodes describing local and global *data* scopes and dashed directed edges representing *memory accesses* to those.

3.1.1 | Control layer

The control layer has two components: function blocks and control transfers. Each function block is described as an address range consisting of the beginning and end instructions. A control transfer edge corresponds to an instruction that can change the active function (call) block. This can be a *direct call*, *indirect call* or *return* instruction, which all make the call-return graph of the programme. Those edges carry address information of target instructions as permitted destinations. Although they already correspond to the beginning of function blocks for call transfers, return destinations are represented by the addresses of call sites. Additionally, for a function block that contains an *indirect jump*, for example, switch statement, the RIM considers a self-referencing edge to the same function block unless the function that contains it is the longjmp function. This is because such instruction cannot branch outside the function in a regular scenario. If there is a non-local jump due to the setjmp/longjmp instance, used for exception handling, the RIM adds an inter-procedural edge from the longjmp function to the function blocks that calls setjmp function. For practicality, the RIM does not represent control transfers at the basic block level, such as unconditional or conditional jumps, the destinations of which are already hard-coded. These control instructions cannot be exploited without modifying the code, which would be an attack scenario that our scheme promises to detect as a code attack, as explained in Section 4. Although hard-coded direct call destination cannot be exploited as well without code corruption first, they are represented by the RIM to track the execution context at the function level.

3.1.2 | Data layer

RIM has an additional layer that describes a coarse-grained model of legitimate memory accesses that must be observed at runtime. This data layer consists of variable groups and

```
1    struct user_info {
2    int user_id;
3    int role_id;
4    int authenticated;
5  } user = { 0, 0, 0 };
6
7  void authenticate(void* func_ptr){
8    char user_name[10];
9    char user_pwd[10];
10   int msg_type;
11   ...
12   /*arbitrary memory write bug*/
13   ...
14   check(user_name,user_pwd);
15   if (user.role_id==2)
16         func_ptr=&priv_session;
17   ...
18   switch(msg_type){
19     ...
20   }
21 }
22
23 void login() {
24     void (*create_session)(int)=&unpriv_session;
25     while (!user.authenticated){
26         authenticate(func_ptr);
27     }
28     (*create_session)(user.user_id);
29     return;
30 }
```

FIGURE 4 Vulnerable programme code that can form a basis to different attack scenarios.

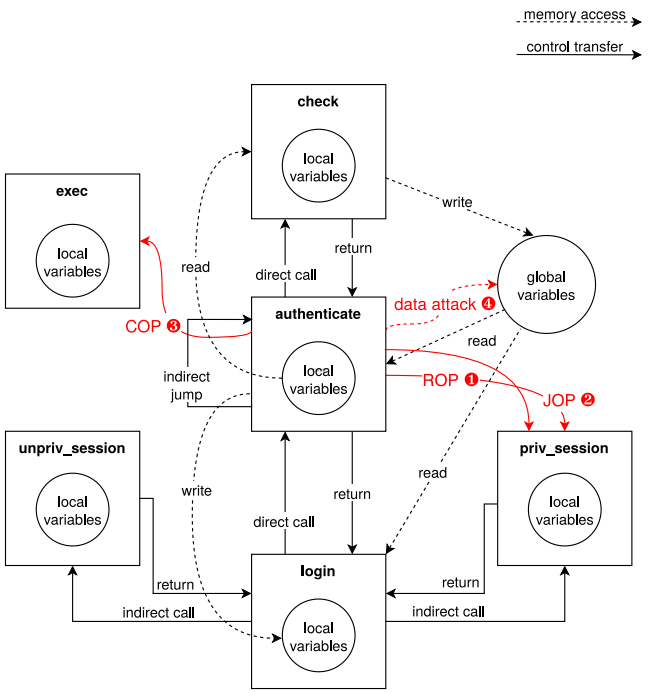

FIGURE 5 Runtime integrity model (RIM) of the vulnerable code in Figure 4 with potential attack scenarios illustrated.

memory accesses. RIM assumes that only the host function can access its local variables (i.e., call frame) unless a variable address is shared as a call by reference argument with a callee function. Second, despite the availability of global variables to the whole programme, the code can statically describe which functions should legitimately access them.

These two layers provide a static approximation of legitimate program executions. Therefore, a code reuse attack that deviates from the expected control flow or violates given data access policies can be detected.

3.2 | Dynamic extensions

Validating programme runtime according to a static model is inevitably subject to over-approximation limitations. More specifically, for a function that can be legitimately called from different (caller) functions, a stateless call graph does not precisely specify the exact function that the callee must return at runtime. An attacker can thus replace the return address of the original (caller) function with another function that the graph permits. A shadow stack (hosting the copy of return addresses) is typically used to differentiate such cases and achieve a more precise return integrity through comparisons of shadow and actual return addresses. However, in the case of recursive functions, a shadow stack cannot be accommodated in a hardware module with limited memory resources. In this work, we extend RIM with *call counters* to attest return addresses with better precision without asking for unbounded memory resources. Each function has a counter value that is set to zero by default. This counter is incremented for a call made from and decremented for a return to that function. Since every caller would be returned in a regular scenario, those counters must be zero at the end of a legitimate programme execution. The verifier can check whether these counters are compliant or not, depending on the last instruction executed. Any inconsistency would reveal attacks that could have stayed unnoticed by a pure graph-based approach.

To illustrate how these counters would enhance the scheme, Figure 6 depicts two synthetic examples with aligned call graphs. At the bottom of each, directed arrows represent call and return instances of their crafted traces. *context* traces describe the current call stack of each programme execution, whereas *expectation* traces show how the executions should complete. Both figures provide example attack traces that pure call-graph-based checks would miss. Specifically, Figure 6a illustrates scenarios that the attacker returns to a different function (e.g., baz ←foo) that is different from the expected one (e.g., baz ←bar). Figure 6b presents a programme with an indirect recursion (e.g., $\mathsf{foo}^1 \rightarrow \mathsf{bar}^1 \rightarrow \mathsf{baz}^1 \rightarrow \mathsf{foo}^2$) where the attacker can return to a different frame context on the stack, which is depicted by the superscript numbers, while skipping some expected returns. Thanks to call counters that keep track of call/return instances made, the tuples provided would reveal these attack scenarios.

4 | RUNTIME MONITORING AND ATTACK DETECTION

With the RIM loaded at deployment time, our hardware security module (HSM) connected to the prover's bus becomes ready to monitor the integrity of both programme code and execution. As depicted in Figure 3, the address bus provides instruction and variable addresses. The bi-directional data bus carries instruction (opcodes and operands) and variable (value) contents from memory, while the control bit indicates the access type. The HSM uses control and address bits to ensure

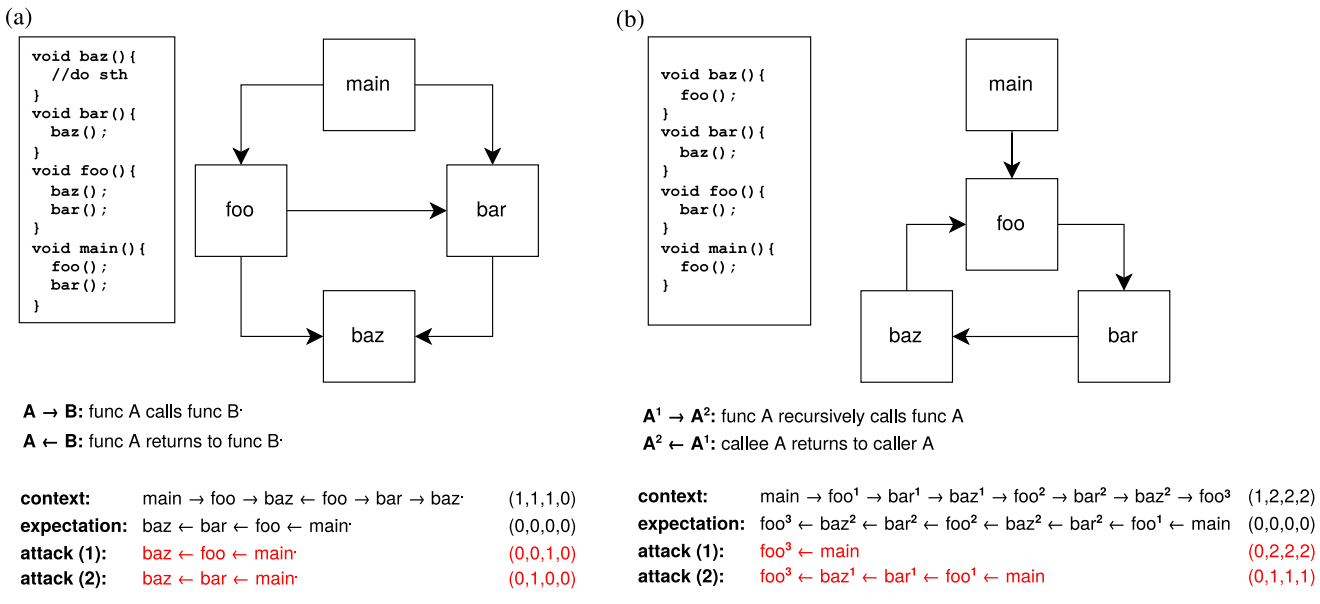

FIGURE 6 Two examples depicting code-reuse attack scenarios that would have stayed undetected without call counters. (a) Programme with functions called by different callers. (b) Programme with functions making indirect recursion.

that programme code is not altered. More importantly, the address bus informs about where control transfers jump to and memory operations access to. On the other hand, data bus values are used to identify what instruction is being fetched. Using those bits as the runtime input and the RIM as a reference model, the HSM measures whether both code and its execution are in a good state.

4.1 | Runtime integrity checks by the HSM

In order to follow up the prover's state, HSM's monitoring logic has six modes, each of which should complete its task in a single bus cycle (see Figure 7). These modes constitute a finite automaton, where each mode corresponds to an automaton state. The HSM starts with *Dispatcher* as the default mode. Depending on the instruction on the bus, this mode causes the HSM to switch to the relevant task mode as the name implies. These task modes are designed to monitor the compliance of control flow transfers and memory accesses with the RIM. If an integrity violation is detected, the HSM sets the appropriate attack flag and stops further monitoring. The HSM has three additional modes to distinguish and report different attacks. *Code attack* implies that the code addresses are illegitimately accessed or corrupted. *Control attack* means a divergence from the expected call-return graph and *Data attack* indicates unexpected memory access to either global data or higher stack addresses (i.e., callers' frames). If the HSM is switched to any of these attack modes, it maintains that state and waits for an attestation request to report the attack details. The verifier needs to perform a hard reset on the HSM to restart the process in a clean state. Figure 8 illustrates the bus cycle-based logic of each mode, of which detailed explanations are given in the following sections.

4.1.1 | Dispatcher

This mode first identifies the range of every address seen on the bus. For an address, pointing data regions, this mode does not take any action and waits for the next bus cycle. In the case of a code address, it first confirms that the control line has a read signal as write access would mean the corruption of the programme code. For read access, this mode identifies the instruction type fetched. If the instruction is one of the control transfer instructions or a store/load operation that needs special treatment, it switches to the appropriate task mode. Otherwise, it maintains the same mode and waits for the next cycle.

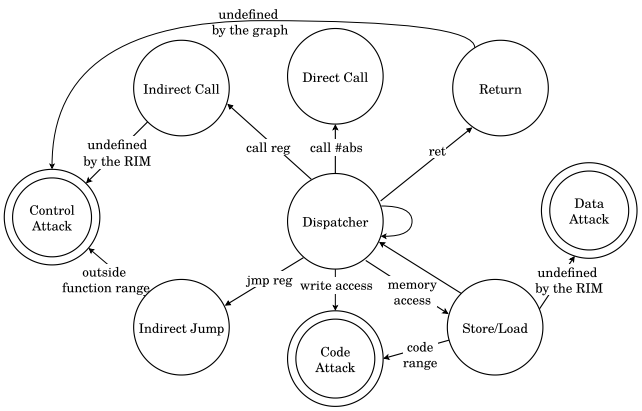

FIGURE 7 Bus-cycle based automaton of Hardware Security Module (HSM) modes.

If a call is made to a hard coded address, the HSM switches to the *Direct Call* mode. In case of an indirect call instruction, whose callee address is given by a register, the HSM mode changes to *Indirect Call*. In contrast, when the instruction is a return instruction as a backward-edge control transfer, the HSM switches to the *Return* mode. If the instruction is an indirect jump, the target of which is not hard-coded, the mode changes to *Indirect Jump*. The HSM does not have a special treatment for direct jumps or any conditional jump instructions since exploiting them requires the code to be altered first. Lastly, if a memory instruction is encountered, the HSM switches to the *Store/Load* mode.

4.1.2 | Direct call

This mode is responsible for keeping track of the execution context on the RIM graph. Following a call instruction, the address in the next bus cycle should be an entry address of a function block known by the RIM (edges). This mode gets the call address on the bus and locates it on the RIM to update the active function node. But prior to the update, it increments the call counter of the function. In addition, this mode handles `setjmp` and `longjmp` calls with special care if the target address belongs to any of these. It stores a copy of the call counters in an array structure within the HSM for a `setjmp` call. Later, this array of call counters is used to update the original counters if a `longjmp` call is encountered.

4.1.3 | Indirect call

This mode works very similar to the previous mode. Differently, it ensures that an indirect call such as a function pointer used is to call a permissible function target, not an arbitrary instruction or a function in the code. It first increments the call counter. Then, it checks whether the address on the bus is a defined edge by the RIM, which is also the first instruction of a permitted function. This is because we cannot allow an indirect call target to be an arbitrary instruction of the target function. Otherwise, the HSM sets the control attack flag if the model does not recognise the destination address.

4.1.4 | Return

The HSM employs this mode to check the integrity of return addresses. When a return instruction is on the bus, this mode checks whether the target address in the next cycle belongs to one of caller sites (i.e., return edges) defined by the RIM. If

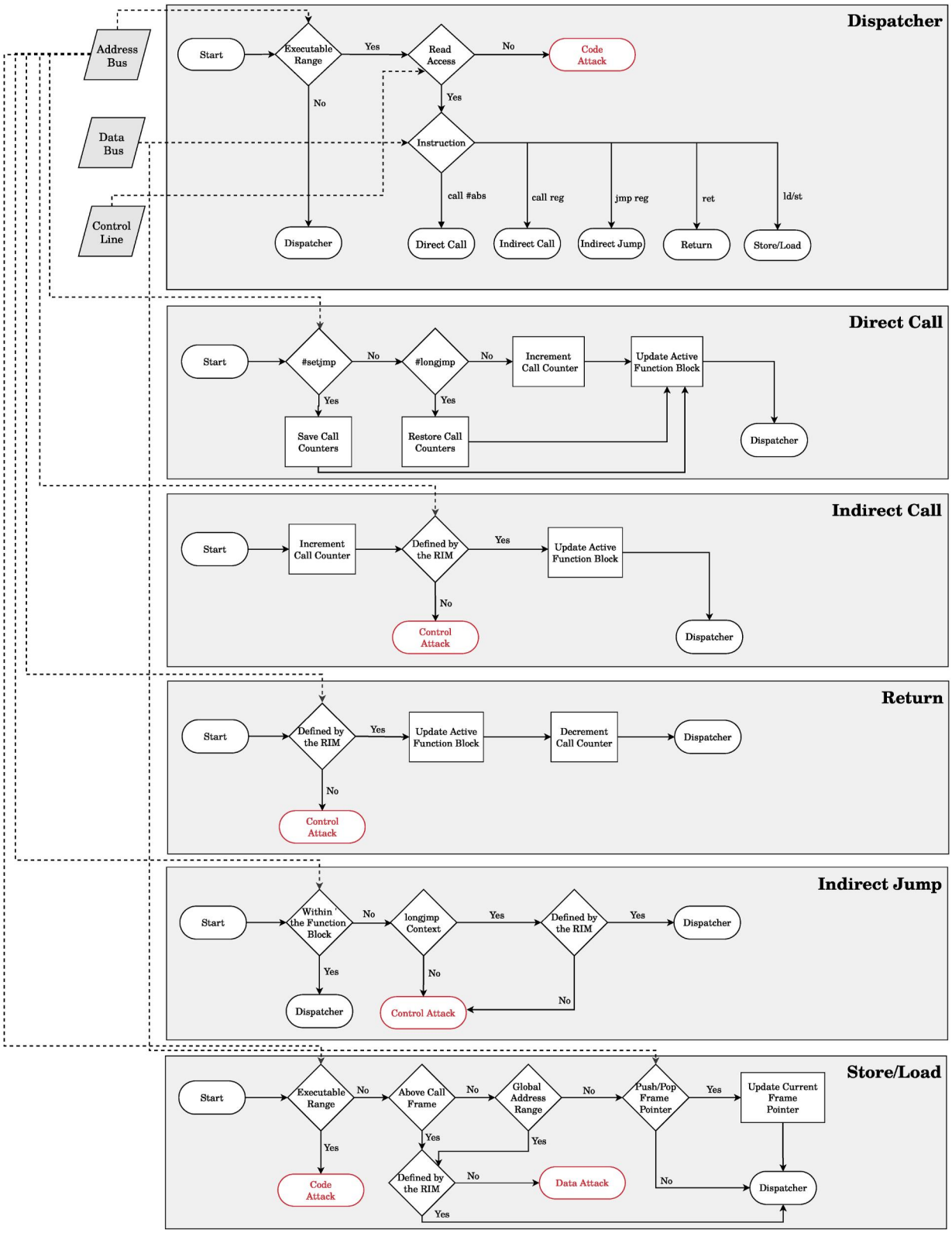

FIGURE 8 Bus cycle-based detail process flow of HSM's monitoring logic using the runtime integrity model (RIM) graph.

not, it sets the control attack flag. Otherwise, it changes the active function context and decrements its call counter.

The call counters mentioned in Section 3.2 are managed by these three modes to achieve more precise return address checks. Since a stateless graph-based approach would not notice the attacker that returns to a different function, these counters aim to approximate the shadow stack precision that would normally require unbounded

memory resources in the presence of different recursive function patterns.

4.1.5 | Indirect jump

The HSM uses this mode to check indirect jumps (e.g., switch statement) that do not link a return. In a regular scenario, we expect jump targets to remain within the existing function. An exception to this would be indirect jumps made by the `longjmp` function. If the active context belongs to the `longjmp` function, it checks whether the target address is one of the `setjmp` sites defined by the RIM. Otherwise, it sets the control attack flag.

We remind that direct jump instructions, both conditional and unconditional, are not monitored, as their targets are given from the code and cannot be exploited without touching the code, which is also attested by the HSM.

4.1.6 | Store/load

This mode performs scope-based checks to report arbitrary memory access attempts. It defines constraints on the address range in which a memory instruction can operate. First, it ensures that the programme code does not write the address range given by the verifier itself, which is necessary to catch code corruption scenarios. We note that legitimate self-mutating code instances are not considered. Therefore, the HSM sets the code attack flag if the operand address of a memory instruction falls within the code range specified at deployment time. Apart from this, the mode follows up two coarse-grained policies defined by the RIM for each function block. It seeks two requirements that must be fulfiled: The first one is that a function without any global/heap variable use should not access non-stack address ranges at runtime. If such function illegitimately overwrites/reads global addresses, the HSM sets the data attack flag. Second, for a function that does not accept any call by reference arguments, all stack accesses must stay within the current call frame; more precisely, accesses above the current frame are described as a data attack during the execution of such a function. To perform this check, the HSM uses the active frame pointer address, which is also extracted by the mode. Because the frame pointer is also saved and restored by a store (push) and load (pop) instruction at function prologues and epilogues, this mode also keeps the copy of the frame pointer within the HSM. For this update, the mode uses the data address accessed during the frame pointer push and the data value read during the pop operation. Although the details can vary depending on the architecture and calling convention in use, it takes the offset of any non-register arguments into account.

4.2 | Attacks coverage

The HSM reveals different attack classes: The first is attacks that corrupt the original programme code. Thanks to *Dispatcher* and *Store/Load* modes, the HSM describes any overwrite of the given code address range using a memory instruction from that range as a *code attack*. This provides strong code integrity attestation for embedded systems that lack architectural and OS support for code and data separation, that is, write-xor-execute (W⊕X). In addition, thanks to continuous monitoring, it promises capturing TOCTOU attacks that would have normally stayed unnoticed between two attestation windows.

The second attack class covered is *control* attacks, such as *code-reuse* and less-sophisticated *code-injection* scenarios, where the primary target of the attacker is control data, such as code pointers. The HSM confirms that any instruction updating the programme counter with a potentially corrupted value sets the counter to a permissible instruction defined by the RIM. This includes both backward-edge return addresses (ROP) and forward-edge targets, such as indirect call (COP) addresses. We do not worry about direct conditional and unconditional jumps since their destination addresses are hard-coded and cannot be altered without a code attack first. To start executing an injected code from the non-code address range or to reuse already existing instructions, the attacker must take over at least a single code pointer. This should eventually cause a divergence from the RIM and will be captured by the HSM as a control attack. For a better reduction of the attack surface, call counters reveal side cases where the attacker crafts return addresses with options that do not diverge from the call-return graph. Despite not being as precise as shadow stacks that preserve the order of calls, call counters significantly reduce the options for the attacker that would be given by a stateless graph. The HSM also checks the constraints described by the RIM for both intraprocedural and interprocedural indirect jumps to reduce useable attack gadgets.

In addition, our scheme considers *data attacks* that reuse the code without altering code pointers. We remind that complete coverage of data attacks normally requires either memory safety or expensive fine-grained data-flow integrity (DFI) checks, which is a non-trivial task to perform with HSM's limited resources. Therefore, the HSM offers only coarse-grained checks. These checks aim to catch accesses to global data by a function without any expected use or accesses to the callers' frames by a function that does not take any reference/pointer arguments. Memory accesses that do not comply with those constraints would thus be reported as a data attack.

5 | PROTOCOL OVERVIEW

This section presents a protocol design that assures that the verifier receives a genuine report through an infected device and an untrusted network. We consider that, at any moment, the verifier can make a request to learn about the prover's internal state. As seen in Figure 9, when the prover receives an attestation request containing a fresh nonce value N generated by the verifier, the prover calls the $Attest(N)$ provided by the HSM's API. Then, the prover needs to send back its output as

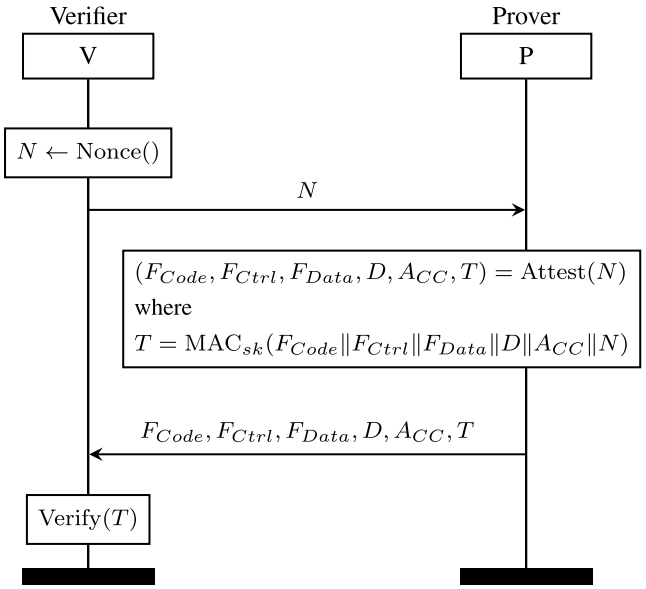

FIGURE 9 Overview of the remote attestation protocol reporting any attack presence on the prover to the verifier.

the attestation response to the verifier. The response consists of code F_{Code}, control F_{Ctrl}, data F_{Data} attack flags, and diagnosis information D about the state prior to the attack, which consists of two registers holding the last executed instruction address and the destination address attempted by a control or memory instruction and the array of call counters A_{CC}. We remind that the HSM stops further monitoring when an attack flag is set. Therefore, the response provides information to the verifier to reason about the instruction exploited and the intended target. Additionally, each response contains a tag T of which all the information and the sent nonce are signed with a MAC scheme. Upon receiving the attestation response, the verifier verifies the tag using the shared key with the HSM. While the key (sk) ensures the authenticity of the message, the tag—digesting nonce N, flags and counters—guarantees the freshness and integrity of the response. The verifier can then check flags and counters to decide about the existence of an attack. F_{Code} flag means a code attack. F_{Ctrl} and F_{Data} flags imply a runtime attack scenario, where the former states a control flow hijack, while the latter tells that there is a data access that violates policies stated by the RIM. Only if all flags are negative and call counters are zero/compliant as expected, the verifier can conclude the prover is in a healthy state.

6 | SECURITY ANALYSIS

To successfully compromise the prover without being detected by the verifier, the adversary must either hide the attack artefacts from the HSM or forge a valid attestation response when queried by the verifier. This section analyses these possibilities with an evaluation of the attacks captured by the RIM on a concrete example.

6.1 | HSM attacks

Due to the system bus integration, every instruction executed and data transferred from/to memory will be monitored. Because physical attacks (e.g., probing) are excluded, any attack has to go through the bus and will be accessible by the HSM. The adversary should modify either the code or its control flow for an attack. Alternatively, the attacker can attempt to find a flaw in the HSM's monitoring logic.

Regarding the first option, if the adversary uses a memory instruction from the given range to modify the code itself, the HSM will report those as code attacks thanks to *Store/Load* mode. Hijacking control flow as a code-injection or -reuse attack is also not practical since the attacker's execution must comply with the RIM. But RIM put constraints on all control instructions, the target addresses of which might reside on dynamic memory regions. *Indirect Call*, *Return* and *Indirect Jump* modes guarantee that the programme counter is always set to an instruction address described by the static model. Additional call counters cover scenarios that might exploit the imperfections of the static return edges. Considering the constraints defined by the *Store/Load* mode, the attacker's ability to manipulate control flow via data attacks is also reduced. As a result, the HSM would catch the attacker for a scenario that does not comply with the RIM.

Regardless of the compliance with the RIM, for an attack targeting HSM's monitoring logic, the adversary must find a bug/flaw that can alter the RIM or dynamic states within the HSM. However, this is unlikely because the monitoring logic implemented as hardware would be free from software vulnerabilities providing too much scope to the attacker with arbitrary read/write capabilities. Thanks to its limited expressiveness and resources, the HSM actually serves as the root of trust on the prover.

6.2 | Protocol attacks

When altering the HSM states is not possible, the only option left to the adversary is to prevent the verifier from seeing the genuine violation flags. To accomplish this, the adversary has to return a valid response to the verifier's request. If the prover does not respond, the verifier will conclude that the prover is compromised. Therefore, the adversary cannot simply block a message or remain silent after compromising the prover. There are only two ways an adversary can send a valid response: either replay a previously captured response or craft one from scratch. We look at each of these in turn.

MAC provided with the response (Figure 9) contains a nonce picked by the verifier. Thus, to replay the response message, the adversary would either have to force the verifier to use the same nonce twice or predict what nonce is going to be used and query the prover ahead of time before compromising the prover to obtain a clean response. This is only possible with negligible probability since we do not allow the adversary to compromise the verifier, and the nonce is chosen securely (i.e., uniformly from a large domain).

Thus, to return a valid message, the adversary must create it from scratch. However, the message must be authenticated using a key kept in the HSM, which the adversary cannot obtain by assumption. Therefore, to forge the message, the attacker has to break the existential unforgeability property of the underlying MAC scheme, which can be done only with negligible probability.

6.3 | A concrete example

This section analyses the effectiveness and limitations of our scheme against attack scenarios that could be performed using the code in Figure 4. A powerful attacker exploiting the arbitrary memory write primitive given in *line 12* would have different options: For example, as a control attack, he can replace the awaiting return address on the stack, which should normally point to the call site at *line 26*, with the address of a different function ❶ such as priv_session. Or he can alter the target address of the indirect jump generated by the switch statement in *lines* 18–20 to perform a jump to any instruction, such as *line 28* or the priv_session function ❷ as a desired outcome. Alternatively, he can corrupt the function pointer defined at *line 24* with the address of a critical system function ❸ (e.g., exec). Also, he can modify global user_info elements defined at *lines 5* as a data attack example ❹ that would help to create a privileged session without any legitimate authentication. For any of these scenarios, the HSM would set the corresponding attack flag as they all constitute a deviation from the RIM graph depicted in Figure 5.

In terms of limitations, we note that RIM cannot approximate all legitimate executions with full precision, like any static programme models. For example, if the attacker replaces the address of unpriv_session with the address of priv_session, the HSM would have to give a pass to such a scenario, as both are valid targets according to the model. Or our scheme does not have much to do if the attacker performs a meaningful attack by exploiting the indirect jump of authenticate while staying within the range of the function. Identifying such attack scenarios is not possible without the knowledge of programme input, which is a known limitation for any static approach. For completeness, we also highlight that our coarse-grained checks on memory accesses leave room for a data attack scenario targeting stack variables at higher frames from a function that has at least a single call-by-reference argument or an attack targeting another local variable within a function. Detection of such data attacks requires more fine-grained checks, such as DFI [16], which cannot be accommodated in a hardware module with very limited resources.

7 | PERFORMANCE

The HSM is designed to perform its checks in real time while the prover keeps running. We remind that the prover has a general-purpose CPU and enough memory resources that can have an unbounded number of call frames. In contrast, the HSM has limited memory and serves a specific purpose, where its hardware is tailored for. For a practical attestation scheme, both the memory usage and the complexity of the HSM tasks should comply with its resource constraints without degrading security guarantees.

In terms of memory requirements, the HSM must provide enough space to host the RIM and call counters. The size of the RIM can be defined as $O(n + e)$, where n is the number of nodes and e is the number of edges in the model. The former corresponds to the number of functions, whereas the latter is mainly defined by the number of call edges from a caller function to distinct functions and the number of return edges to different call sites. We note that both (intraprocedural) indirect jump and memory access edges illustrated in Figure 5 do not scale per function nor increase the complexity of the RIM since their checks are not address-specific. Hence, the number of functions and call-return relations between them represents the main cost of the static part. For the dynamic part, the space required by call counters is also defined by the number of programme functions, regardless of the depth or recursiveness the call stack might have at runtime. Despite being programme-specific, we can approximate the memory requirement of a RIM as a function of the programme size. To provide insight into such evaluation, we have analysed three bare-metal examples of different sizes. Those binaries are JTAG, bootloader and compression library implementations with components, including UART, Adler, CRC32 checksums and memory allocators. Table 1 summarises the number of instructions and key RIM components found in those instances. We highlight that RIM, centred around the programme's call-return graph, provides a more succinct representation of the binaries with smaller sizes. For instance, zlib, as the most complex example consisting of more than 9 K instructions, is modelled using a far less number of components with 57 function blocks (address ranges) and an average of 1.4 call and 2.8 return (address) edges per function. With an average of 14% model size to binary size ratio, RIM requires reasonable memory resources.

Regarding the complexity of HSM tasks, each depicted mode has a different process flow. Many modes, such as *Dispatcher*, *Indirect Jump* and *Store/Load*, fulfil their tasks within constant time. On the other hand, *Direct Call*, *Indirect Call* and *Return* modes perform a linear search task whose cost is normally defined by the degree of the active function node in the RIM. However, those searches are expected to be bounded in practice due to the limited number of functions. For instance, *zlib*, as the most complex example examined, does not include any function block with more than eight call edges as shown in Figure 10. Therefore, those searches can be parallelised at the hardware level with a small content-addressable memory buffer that would host the data of the active function block and complete the search in constant time. We emphasise that each mode intends to complete its task within the same bus cycle. To perform these tasks in real time, we consider a non-generic hardware-based implementation, such as FPGA, for the monitoring logic described in Figure 8.

TABLE 1 Instruction counts and the complexity of runtime integrity models (RIM) of bare-metal instances.

	Instructions			RIM highlights per function				
	Count	Call [%]	Ret [%]	Functions	Avg-call	Avg-ret	Max-call	Max-ret
Jtag	165	12.1	1.8	5	0.8	3.3	3	10
Bootloader	554	12.8	2.7	17	1.3	3.4	6	8
Zlib	9068	2.6	1.3	57	1.4	2.7	8	19

Implementation of the monitoring logic at a lower abstraction layer would enable the HSM to process much faster, where the entire process of each bus mode can be completed in a single tick of the FPGA's clock.

8 | RELATED WORK

8.1 | Static attestation

Software-based attestation schemes, such as SWATT [17] and Pioneer [7], suggest calculating checksums via pseudorandom memory traversals. Because the traversals are initiated by the nonce given by the verifier, a delayed response of the prover would be interpreted as an attack. Although those schemes do not require a key, their correctness is dependent on how time-optimal checksum calculations are. On the other hand, hardware-based schemes can address stronger adversaries with reliance on trust anchors, such as trusted platform modules [18, 19] and physically unclonable functions [20]. There are also hybrid approaches [21–23] that leverage more commonly available hardware components, such as read-only memory (ROM) and memory protection units. For instance, SMART [21] suggests storing the key and the attestation code on ROM with additional control mechanisms on the bus to ensure that only the attestation code can access the key. A relevant work [24] describes security requirements for the attestation key and attestation code. They suggest two properties for the attestation key such as *exclusive access* of the attestation code with an assurance of *no leakage* afterwards and three features for attestation code, which are *immutability*, *interruptibility* and *controlled invocation* as an enforcement of a legitimate entry point to the code. In a recent work, VRased [25] extends MCUs with a hardware module that can similarly obtain information about the instructions and memory addresses accessed. Differently from our work, the information is used to verify the security properties of the static attestation process. RATA [1] elegantly addresses TOCTOU attack scenarios targeting static regions. RATA logs the modification time of those memory regions in a fixed memory address using VRased, whenever these regions are touched, and includes this information in the checksum provided as part of attestation responses. Therefore, the verifier can confirm that the contents of attested regions are not changed at a time unknown to the verifier. Unfortunately, RATA and similar checksum-based static attestation methods do not cover attack scenarios that reside in dynamic memory regions, which are addressed by runtime attestation schemes.

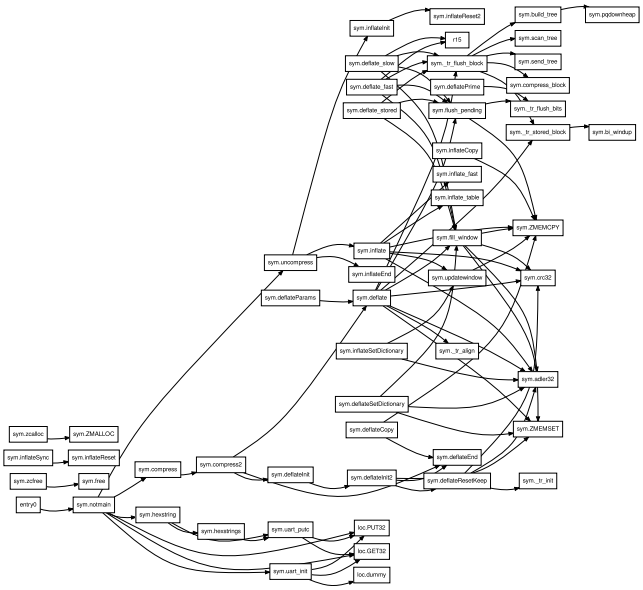

FIGURE 10 Call graph of *zlib* library as the most complex bare-metal software evaluated.

8.2 | Runtime attestation

DynIMA [26] extends static attestation [18] by introducing a taint tracking mechanism for untrusted data used as code pointers. ReDAS [27] adopts a different approach that checks the fulfilment of two kinds of system properties. These are formulated as *structural integrity*, for example, any return address has to be the instruction address following its call instruction, and *global data integrity* constraints, for example, data invariants. Both schemes are designed for high-end devices requiring operating systems. An increasing number of studies in recent years promises control-flow attestation for more constrained embedded devices. For example, both C-FLAT [2] and LO-FAT [3] propose digesting path traces on the prover side and providing a single cumulative hash to the verifier to provide the information of executed branches. C-FLAT uses binary instrumentation and TrustZone for this purpose, whereas LO-FAT suggests extending hardware for branch recording and loop monitoring. Both studies aim to reveal control-flow hijack and some of the data attacks, such as control-flow bending scenarios. But to reason about provided hash values, the verifier must discover all extractable path traces using the CFG of the programme. However, the search space can explode quickly for many programs with enough number of nested control structures and indirect calls, such as

zlib. Furthermore, even if all CFG paths are discovered, deciding whether a hash (i.e., path trace) represents a data (control-flow bending) attack requires the knowledge of external programme input. We remind that in a system setting [3] where the verifier has the programme and also defines its input—which should typically be determined by the environment the prover is in—actual computation task assigned to the prover could have been performed on the verifier side as the trusted party. Apart from the schemes digesting control-flow events, more recent schemes [28, 29] offer data-flow attestation mechanisms. For instance, LiteHAX [28] sends path traces as bitstream and creates a cumulative hash for only memory (store/load) traces this time. So the verifier needs to discover legitimate memory accesses for the execution path given by the bitstream traces. Because it has to provide control-flow information (trace) to the verifier periodically in a lossless way, LiteHAX yields more communication overhead. In addition, two schemes, Tiny-CFA [30] and DIALED [31], relying on the VRased [25] architecture suggest logging control-flow events and external inputs with the proof-of-execution model of APEX [32]. The former aims to reveal control attacks, while the latter can be used to detect data attacks. In general, existing runtime attestation methods either log or digest runtime events to provide information about what happened on the prover side. The verifier can accordingly check their correctness later. However, the logging-based approach introduces space and communication overheads, whereas digest-based approaches require searching in the state space that can quickly explode. Therefore, both approaches do not scale easily to more complex software instances.

To address these challenges by offloading some of the checks to the resources that are already required for logging or digesting, the conference version of this work [5] suggested a hardware security module that is similarly connected to prover's system bus. However, the static runtime model and the HSM logic were completely different in the conference version with further limiting assumptions on the software attested, such as the use of recursive functions and hard precision requirements for the static model. Unlike the conference version adopting a branch-centric (CFG) runtime model, this paper uses a more lightweight call-centric model that describes the expected programme flow through function calls with also less fine-grained checks on data accesses. Furthermore, the hardware logic in the conference version mainly identifies instructions via model-given addresses on the bus, whereas the HSM in this paper adopts an opcode-based monitoring logic using data bus values. Lastly, unlike the conference version asking for a shadow stack, this paper employs call-depth counters and suggests a more practical scheme that can attest software instances even with recursive function calls.

8.3 | Exploit mitigation

Apart from attestation schemes, many mitigation techniques are also proposed in the literature against runtime attacks. Differently, these studies prevent an attack state from happening in the first instance. For example, the seminal control flow integrity work [33] ensures that execution control is always transferred to an address defined by the programme's CFG. Code pointer integrity [34] suggests deploying a safe stack to protect code pointers directly from memory corruptions instead of validating their address values. The correctness of those schemes relies on the integrity of (instrumented) code and instrumentation data that must be kept on the same memory space. Hardware-based solutions, such as HCFI [35] and HAFIX [36], modify instruction sets (ISA). Despite their benefits over software-based solutions, such as stronger protection and less overhead, their on-chip design increases deployment costs for existing devices with legacy issues. On the other hand, against data attacks, a better approximation of memory safety, DFI [16] uses *reaching definitions* analysis to mitigate both control and non-control data attacks. Data-flow integrity maintains a runtime definitions table, which logs defining (write instructions) on each memory address to later check whether they are written by expected instructions. Hardware-based data-flow isolation, HDFI [37], proposes a similar but more coarse-grained approach. Instead of checking instruction identifiers, HDFI splits memory addresses as sensitive and non-sensitive via one-bit tags. In general, most of these mitigation techniques are available to high-end systems and used in settings where the termination of programme execution does not constitute an attack.

9 | DISCUSSION

9.1 | Possible extensions

Despite being orthogonal to this study, in case the prover has a remote update mechanism for its firmware, the HSM API can also be extended to handle the update of a matching RIM remotely and securely. Because the HSM and the verifier already share a key, an extended HSM and a protocol design can verify both the authenticity and integrity of the new RIM received. We note that the monitoring logic implemented as hardware can still operate on different RIM instances kept as data in the HSM's memory and therefore can be updated remotely with the necessary changes.

9.2 | Cost considerations

Like any security mechanism, our attestation brings additional costs due to a requirement of the FPGA. Such device cost is justifiable for many critical domains in automotive, health, and military systems that require high-integrity assurance. In the case of a sensor network, HSM devices can be selectively integrated into a few cluster heads. For other use cases, we remind that the idea of integrating FPGAs into low-cost MCU systems is not new. Those programmable logic units are commonly used in embedded space to take over some workloads that cannot be efficiently processed by MCUs. For instance, many battery-powered wireless sensors [38, 39]

benefit from FPGAs to accelerate their computation-heavy tasks, while maintaining low-cost and low-power characteristics of the system. Furthermore, embedded platforms such as discontinued AT91CAP7 series could provide native MCU-and-FPGA integration as a complete solution with metal programmable cell fabric [40] and faster communication interfaces [41]. Due to economies of scale, such devices with promising costs of $6–13 [42] for each in large quantities (100K) could be used to bring our security promises to non-critical domains with easily justifiable costs.

9.3 | Energy estimations

The energy consumption of an FPGA-based HSM would be very specific to the software task running on the prover's platform. This is because the nature of prover's task and its environment define how active or idle the prover's bus, and so the HSM should be at runtime. Therefore, actual CPU time required to perform the task is crucial for energy costs since both the prover (MCU) and HSM (FPGA) would save significant energy by putting themselves into low-power sleep modes during their idle modes. For instance, many MCUs operating at low voltages (1–5 V) employ different energy modes that range from a light sleep (45–200 μA/MHz), standby mode (1–50 μA) to deep-sleep (20–400 nA) [43]. Similarly, FPGAs are not necessarily active all the time and can spend most of its time in standby and sleep modes, therefore can save significant energy. Especially, flash-based FPGAs, such as IGLOO series [44], require less static power to preserve its state compared to RAM-based options and can promise much lower consumptions during these idle times with a power consumption ranging from 5 μW per 15 K gates to 53 μW for 1 M gates. Apart from the static power usage, FPGAs have reasonable dynamic power consumption ranging from 5 mW to 20 mW [45]. We remind that a typical MCU (e.g., AT32UC3A0512) could be a more power-hungry while actively operating at 66 MHz with a reported of 40 mA at 3.3 V [46].

10 | CONCLUSION

This paper presents a novel remote attestation scheme that addresses both code and execution integrity of embedded systems responsible for critical tasks. The scheme employs a non-intrusive hardware security module (HSM) loaded with a static runtime integrity model of the programme. Thanks to HSM's continuous monitoring through the prover's bus, our scheme reveals any attempts to corrupt or replace the genuine programme code. Unlike conventional static attestation schemes, this provides immunity to TOCTOU scenarios exploiting temporal gaps between two measurements for code replacement and corruption.

Apart from the code attestation, our scheme also attests how the legitimate code is executed considering its both control and data features. Hence, without having to discover all possible execution paths in advance—-subject to path explosion—or introducing significant overheads by logging them, our scheme reports both code and code-reuse attacks that can target the prover. The proposed HSM design with system bus integration not only makes our scheme compliant for critical embedded systems with legacy issues but also monitors the prover's execution from a point that the adversary cannot hide without a physical attack.

AUTHOR CONTRIBUTIONS

Munir Geden: Conceptualization, methodology, validation, visualization, writing – original draft. **Kasper Rasmussen**: Project administration, supervision, writing – review and editing.

ACKNOWLEDGEMENTS

Munir Geden's doctoral studies were supported by the Ministry of National Education in the Republic of Turkey during the preparation of the conference paper [5], on which this work is based.

CONFLICT OF INTEREST STATEMENT

None.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this as no new data were created. The bare metal examples analysed in Section 7 can be found at https://github.com/dwelch67/raspberrypi.

ORCID

Munir Geden https://orcid.org/0000-0001-6086-0772

REFERENCES

1. De Oliveira Nunes, I., et al.: On the TOCTOU problem in remote attestation. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 2921–2936. ACM, New York (2021). ISBN 9781450384544. https://doi.org/10.1145/3460120.3484532
2. Abera, T., et al.: Andrew paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 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. ACM, New York (2016). ISBN 9781450341394. https://doi.org/10.1145/2976749.2978358
3. Dessouky, G., et al.: LO-FAT: low-overhead control flow ATtestation in hardware. In: Proceedings of the 54th Annual Design Automation Conference 2017, pp. 1–6. ACM, New York (2017). ISBN 97814 50349277. https://doi.org/10.1145/3061639.3062276
4. Zeitouni, S., et al.: ATRIUM: runtime attestation resilient under memory attacks. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), vol. 11, pp. 384–391. IEEE (2017). ISBN 978-1-5386-3093-8. https://doi.org/10.1109/ICCAD.2017.8203803
5. Geden, M., Rasmussen, K.: Hardware-assisted remote runtime attestation for critical embedded systems. In: 2019 17th International Conference on Privacy, Security and Trust (PST), vol. 8, pp. 1–10. IEEE (2019). ISBN 978-1-7281-3265-5. https://doi.org/10.1109/PST47121.2019.8949036
6. Wurster, G., van Oorschot, P.C., Somayaji, A.: A generic attack on checksumming-based software tamper resistance. In: 2005 IEEE Symposium on Security and Privacy (S& P'05), pp. 127–138. IEEE (2005). ISBN 0-7695-2339-0. https://doi.org/10.1109/SP.2005.2
7. Seshadri, A., et al.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles - SOSP

'05, vol. 39, p. 1. ACM Press, New York (2005). ISBN 1595930795. https://doi.org/10.1145/1095810.1095812
8. Claude, C., et al.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of the 16th ACM Conference on Computer and Communications Security - CCS '09, p. 400. ACM Press, New York (2009). ISBN 9781605588940. https://doi.org/10.1145/1653662.1653711
9. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without Function Calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security - CCS '07, p. 552. ACM Press, New York (2007). ISBN 9781595937032. https://doi.org/10.1145/1315245.1315313
10. Buchanan, E., et al.: When good instructions go bad: Generalizing Return-Oriented Programming to RISC. In: Proceedings of the 15th ACM Conference on Computer and Communications Security - CCS '08, p. 27. ACM Press, New York (2008). ISBN 9781595938107. https://doi.org/10.1145/1455770.1455776
11. Stephen, C., et al.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security - CCS '10, p. 559. ACM Press, New York (2010). ISBN 9781450302456. https://doi.org/10.1145/1866307.1866370
12. Tyler, B., et al.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11, p. 30. ACM Press, New York (2011). ISBN 9781450305648. https://doi.org/10.1145/1966913.1966919
13. Chen, S., et al.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium, vol. 5, pp. 146. USENIX Association (2005)
14. Hu, H., et al.: Data-oriented programming: on the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP), vol. 5, pp. 969–986. IEEE (2016). ISBN 978-1-5090-0824-7. https://doi.org/10.1109/SP.2016.62
15. Ispoglou, K.K., et al.: Block oriented programming: automating data-only attacks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1868–1882. ACM, New York (2018). ISBN 9781450356930. https://doi.org/10.1145/3243734.3243739
16. Castro, M., Costa, M., Tim Harris: Securing software by enforcing data-flow integrity. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation - USENIX Association, pp. 147–160. USENIX Association (2006). ISBN 1931971471
17. Seshadri, A., et al.: SWATT: software-based attestation for embedded devices. In: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, pp. 272–282. IEEE (2004). ISBN 0-7695-2136-3. https://doi.org/10.1109/SECPRI.2004.1301329
18. Sailer, R., et al.: Design and implementation of a TCG-based integrity measurement architecture. In: USENIX Security Symposium, vol. 13, pp. 223–238 (2004). https://doi.org/10.1109/MSP.2010.92
19. Tan, H., Tsudik, G., Jha, S.: MTRA: multiple-tier remote attestation in IoT networks. In 2017 IEEE Conference on Communications and Network Security (CNS), Vol. 10, pp. 1–9. IEEE (2017). ISBN 978-1-5386-0683-4. https://doi.org/10.1109/CNS.2017.8228638
20. Kong, J., et al.: PUFatt: embedded platform attestation based on novel processor-based PUFs. In: Proceedings of the the 51st Annual Design Automation Conference on Design Automation Conference - DAC '14, pp. 1–6. ACM Press, New York (2014). ISBN 9781450327305. https://doi.org/10.1145/2593069.2593192
21. El Defrawy, K., et al.: SMART: secure and minimal architecture for (establishing dynamic) root of trust. In: Ndss, vol. 12, pp. 1–15 (2012)
22. Koeberl, P., et al.: TrustLite. In: Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14, pp. 1–14. ACM Press, New York (2014). ISBN 9781450327046. https://doi.org/10.1145/2592798.2592824
23. Brasser, F., et al.: TyTAN: Tiny trust anchor for tiny devices. In: Proceedings of the 52nd Annual Design Automation Conference, pp. 1–6. ACM, New York (2015). ISBN 9781450335201. https://doi.org/10.1145/2744769.2744922
24. Francillon, A., et al.: A minimalist approach to remote attestation. In: Conference on Design, Automation & Test in Europe - DATE'14, pp. 1–6. IEEE (2014). ISBN 978-3-9815370-2-4. https://doi.org/10.7873/DATE.2014.257
25. De Oliveira Nunes, I., et al.: VRased: a verified hardware/software co-design for remote attestation. In: Proceedings of the 28th USENIX Security Symposium, pp. 1429–1446 (2019). ISBN 9781939133069
26. Lucas, D., Sadeghi, A.-R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing - STC '09. ACM Press, New York (2009). ISBN 978 1605587882. https://doi.org/10.1145/1655108.1655117.49
27. Kil, C., et al.: Remote attestation to dynamic system properties: towards providing complete system integrity evidence. In: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks, vol. 6, pp. 115–124. IEEE (2009). ISBN 978-1-4244-4422-9. https://doi.org/10.1109/DSN.2009.5270348
28. Dessouky, G., et al.: LiteHAX: lightweight hardware-assisted attestation of program execution. In: Proceedings of the International Conference on Computer-Aided Design - ICCAD '18 (2018). ISBN 978145 0359504
29. Kuang, B., et al.: DO-RA: data-oriented runtime attestation for IoT devices. Comput. Secur. 97, 101945 (2020). ISSN 01674048. https://doi.org/10.1016/j.cose.2020.101945
30. De Oliveira Nunes, I., Jakkamsetti, S., Tsudik, G.: Tiny-CFA: minimalistic control-flow attestation using verified proofs of execution. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), vol. 2021-Febru, pp. 641–646. IEEE (2021). ISBN 978-3-9819263-5-4. https://doi.org/10.23919/DATE51398.2021.9474029
31. De Oliveira Nunes, I., Jakkamsetti, S., Tsudik, G.: DIALED: data integrity attestation for low-end embedded devices. In: 2021 58th ACM/IEEE Design Automation Conference (DAC), Volume 2021-Decem, vol. 12, pp. 313–318. IEEE (2021). 978-1-6654-3274-0. https://doi.org/10.1109/DAC18074.2021.9586180
32. De Oliveira Nunes, I., et al.: APEX: a verified architecture for proofs of execution on remote devices under full software compromise. In: Proceedings of the 29th USENIX Security Symposium, pp. 771–788. USENIX Association (2020). ISBN 9781939133175
33. Abadi, M., et al.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security - CCS '05, p. 340. ACM Press, New York (2005). ISBN 1595932267. https://doi.org/10.1145/1102120.1102165
34. Kuznetsov, V., Szekeres, L., Payer, M.: Code-pointer integrity. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, Number October, pp. 147–163 (2014). ISBN 9781931971164
35. Christoulakis, N., et al.: HCFI: hardware-enforced control-flow integrity. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 38–49. ACM, New York (2016). ISBN 978 1450339353. https://doi.org/10.1145/2857705.2857722
36. Lucas, D., et al.: HAFIX: hardware-assisted flow integrity extension. In: Proceedings of the 52nd Annual Design Automation Conference, pp. 1–6. ACM, New York (2015). ISBN 9781450335201. https://doi.org/10.1145/2744769.2744847
37. Song, C., et al.: HDFI: hardware-assisted data-flow isolation. In: 2016 IEEE Symposium on Security and Privacy (SP), vol. 5, pp. 1–17. IEEE (2016). ISBN 978-1-5090-0824-7. https://doi.org/10.1109/SP.2016.9
38. Grassi, P.R., Sciuto, D.: Energy-aware FPGA-based architecture for wireless sensor networks. In: Proceedings - 15th Euromicro Conference on Digital System Design, DSD 2012, pp. 866–873 (2012). https://doi.org/10.1109/DSD.2012.50
39. Yamaguchi, S., et al.: Programmable wireless sensor node featuring low-power FPGA and microcontroller. In: 2013 International Joint Conference on Awareness Science and Technology & Ubi-Media Computing (iCAST 2013 & UMEDIA 2013), vol. 11, pp. 596–601. IEEE (2013). ISBN 978-1-4799-2364-9. https://doi.org/10.1109/ICAwST.2013.6765509

40. Atmel: Customizable Microcontroller AT91CAP7S450A, AT91CAP7S 250A. URL https://pdf1.alldatasheet.com/datasheet-pdf/view/255445/ATMEL/AT91CAP7S.450A.html
41. Atmel: Customizable Microcontroller AT91CAP7E. URL https://pdf1.alldatasheet.com/datasheet-pdf/view/257048/ATMEL/AT91CAP7E.html
42. Atmel Announces CAP Customizable Microcontrollers | Berkeley Design Technology, Inc. URL https://www.bdti.com/InsideDSP/2007/07/18/atmel-announces-cap-customizable-microcontrollers
43. Understanding MCU sleep modes and energy savings - Embedded.com. URL https://www.embedded.com/understanding-mcu-sleep-modes-and-energy-savings/
44. Low Power Fpgas: IGLOO Series Industry's Lowest Power FPGAs ProASIC3 Series. URL www.actel.com.cn
45. Actel. Total System Power - Evaluating the Power Profile of FPGAs. Technical report, (2008). URL https://www.microsemi.com/document-portal/doc_view/125155-total-system-power-brochure
46. AT32UC3A0512: Microchip Technology. URL https://www.microchip.com/en-us/product/AT32UC3A0512

How to cite this article: Geden, M., Rasmussen, K.: Hardware-assisted remote attestation design for critical embedded systems. IET Inf. Secur. 17(3), 518–533 (2023). https://doi.org/10.1049/ise2.12113