ATRIUM: Runtime Attestation Resilient Under Memory Attacks

Shaza Zeitouni
TU Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

Ghada Dessouky
TU Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Orlando Arias
University of Central Florida, USA
oarias@knights.ucf.edu

Dean Sullivan
University of Florida, USA
deanms@ufl.edu

Ahmad Ibrahim
TU Darmstadt, Germany
ahmad.ibrahim@trust.tu-darmstadt.de

Yier Jin
University of Florida, USA
yier.jin@ece.ufl.edu

Ahmad-Reza Sadeghi
TU Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Remote attestation is an important security service that allows a trusted party (verifier) to verify the integrity of a software running on a remote and potentially compromised device (prover). The security of existing remote attestation schemes relies on the assumption that attacks are *software-only* and that the prover's code cannot be modified at runtime. However, in practice, these schemes can be bypassed in a stronger and more realistic adversary model that is hereby capable of controlling and modifying code memory to attest benign code but execute malicious code instead – leaving the underlying system vulnerable to Time of Check Time of Use (TOCTOU) attacks.**

In this work, we first demonstrate TOCTOU attacks on recently proposed attestation schemes by exploiting physical access to prover's memory. Then we present the design and proof-of-concept implementation of ATRIUM, a *runtime* remote attestation system that securely attests both the code's binary and its execution behavior under memory attacks. ATRIUM provides resilience against both software- and hardware-based TOCTOU attacks, while incurring minimal area and performance overhead.

***Index Terms*—Attestation, runtime, memory attacks**

I. Introduction

Recent high-profile attacks on embedded systems, such as Mirai and Stuxnet, have become crucially alarming and of increased significance as systems are becoming more interconnected and collaborative. *Remote attestation* plays an important role as a security service for detecting malware on a remote device. It is implemented as a challenge-response protocol that allows a trusted *verifier* to obtain an authentic report about the (software) state of a potentially untrusted remote device called *prover*. Conventional attestation schemes are static in nature, i.e., the prover sends an authenticated report to the verifier by issuing a digital signature or cryptographic MAC (Message Authentication Code) over the verifier's challenge and the *measurement* (typically hash) of the binary code to be attested [22]. However, static attestation only ensures the integrity of binaries but *not* of their execution. In particular, it cannot detect the prevalent state-of-the-art runtime attacks that do not modify the program binary but subvert the intended control flow of the targeted application program during its execution. Current runtime attacks take advantage of code-reuse techniques, such as return-oriented programming that dynamically generate malicious code by chaining together code snippets (called gadgets) of benign code *without* requiring to inject any malicious code/instructions [24]. Consequently, the hash value computed over the binaries remain unchanged and the attestation protocol succeeds, although the prover has been compromised. These sophisticated exploitation techniques have been shown effective on many processor architectures, such as Intel x86 [23], SPARC [4], ARM [16], and Atmel AVR [10]. In fact, large-scale investigations of embedded systems security have shown various vulnerabilities, including memory corruption (such as buffer overflow) that can be exploited for runtime attacks.

Hence, effective attestation should enable reporting the prover's dynamic behavior – more concretely, its current execution details – to the verifier. To attest the dynamic program behavior researchers have proposed enhancements and/or extensions to static binary attestation (e.g., [11], [3]). The most recent, C-FLAT [3], reports the prover's dynamic state (execution paths) and provides fine-grained control-flow measurements to the verifier. Note that, unlike control-flow integrity (CFI) enforcement, control-flow attestation provides detailed information about the executed path that might be of crucial interest to a remote verifier. This information helps in detecting data-oriented non-control attacks [5] that can bypass CFI by corrupting data variables to execute a valid but unintended control-flow path, for instance, redirecting the control flow to a high-privileged recovery routine (see also [13]). However, C-FLAT requires program code instrumentation and incurs high performance overhead, particularly on the prover.

On the other hand, all existing attestation schemes (including C-FLAT) rule out physical attacks in their adversary model. This assumption is not always realistic, since the adversary may at some point have physical access to the prover. In this case, it is possible to execute (extraordinarily effective and cheap) non-invasive attacks on the program code memory through *physical access*. In particular, the adversary physically controls and modifies the memory such that benign code is attested but malicious code is executed instead.

Goals and Contributions. In this paper, we first demonstrate that – using external interfacing with prover's program code memory bank – an adversary can bypass all existing attestation schemes and deliver sound attestation reports, without even having to extract the prover's secret keys (cf. § III). To overcome the limitations of current attestation schemes, we introduce a holistic approach to attestation ATRIUM, a *resilient runtime attestation* scheme that is capable of detecting both physical memory attacks and software attacks including runtime attacks by attesting the executed instructions and their control flow at runtime. Our main contributions are listed as follows.

- We demonstrate memory bank attacks on state-of-the-art attestation schemes for embedded devices such as SMART [9] and C-FLAT [3]. We exploit physical access to code memory to bypass attestation and deliver sound attestation reports without having to extract the prover's secret keys.
- We present ATRIUM– an attestation scheme which: (1) detects memory bank attacks by attesting instructions as they are fetched from (off-chip) memory for execution; (2) prevents software attacks on the attestation process itself by separating the attestation engine from the processor (i.e., no instructions are sent to the processor to perform attestation). Instead, attestation is performed by a separate hardware engine in parallel. (3) detects runtime attacks by tracking and reporting both executed instructions and control-flow events during execution.
- We present a proof-of-concept implementation and performance analysis which demonstrate the effectiveness and feasibility of ATRIUM, and its applicability to low-end embedded devices.

II. Background

Control-Flow Graph (CFG). The execution flow of a program can be abstracted into a control-flow graph (CFG) by leveraging the aid of static and dynamic code analysis. The nodes in CFG represents basic blocks of a program, while edges represent control-flow transitions from one block to another by means of a branch instruction. A *valid* path in CFG is composed of several nodes connected by edges.

Runtime Attacks. An outline of the different classes of runtime attacks is illustrated in Figure 1. The system dedicates separate memories for data and code. The former is marked as readable and writable *(rw)*, while the latter is marked as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten *by means of software*. Along this CFG, we can outline three major classes of runtime attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop variables, and ❸ code-pointer overwrite attacks. By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses and function pointers) as in ❸ an attacker can redirect the control flow of a program such that execution has a malicious and unauthorized effect. In attacks based on *code-injection*,

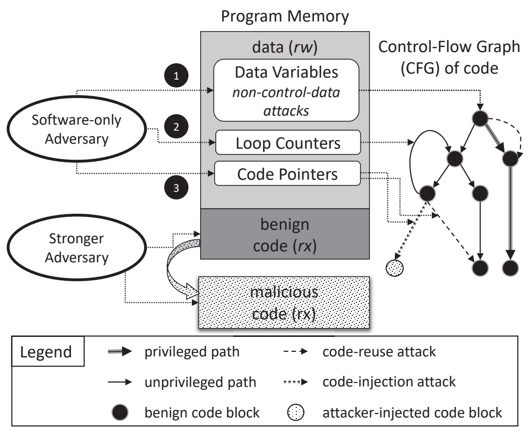


Figure 1: Different attack classes

the attacker places a malicious executable payload in program memory and redirects control flow to execute it. Alternatively, state-of-the-art runtime attacks leverage *code-reuse* techniques, such as *Return-oriented Programming* (ROP) [23]. These attacks exploit a memory corruption vulnerabilities (e.g., buffer overflows) in the program and stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the memory of the vulnerable program. *Non-control-data attacks* [5] do not compromise the control flow of a program, but cause unexpected malicious control flow by corrupting critical data variables such as an authentication variable. This results in executing a privileged (unintended) but permissible control-flow path that exists in the CFG. Attack ❷ affects the number of times a program loop executes by corrupting a loop variable such as a counter. This can have severe consequences depending on the context, e.g., a syringe pump dispenses more liquid than requested (see [3]). Code injection attacks can be prevented by either marking memory as writable or executable. This mechanism is known as *Data Execution Prevention* (DEP) [12]. Countermeasures against code reuse attacks include: *Control-Flow Integrity* (CFI) [2], fine-grained code randomization [19], and Code-Pointer Integrity (CPI) [18].

Besides software-based runtime attacks, a stronger adversary as shown in Figure 1, can modify program code in memory through *physical access* without mounting sophisticated invasive physical attacks, but by simply replacing the benign code memory with malicious code memory at runtime. We elaborate on these memory bank attacks next in § III and propose an attestation scheme that can mitigate them in § V.

III. TOCTOU Attacks on Attestation Schemes

Next we describe memory bank attacks that we aim to mitigate in this work, and we show how they bypass recently proposed attestation schemes: SMART [9] C-FLAT [3] and LO-FAT [7]. These attacks assume a stronger adversary that can physically manipulate the code memory without the need for sophisticated invasive physical attacks and can consequently bypass attestation schemes that strictly consider software-only adversary. The attack is illustrated in Figure 2: At $\mathcal{P}rv$'s side the attestation scheme (i.e., the attestation code and secret key) is stored on-chip while the benign code resides in an external memory. The adversary can interleave instruction fetches to malicious code in-between those fetches needed to attest the benign code of the original program. This can be done by replacing the original memory interface with an interface to a memory controller. This allows the adversary to direct instruction fetches to either benign code when attestation is running, or malicious code otherwise. The same interleaving attack can be achieved by inserting malicious instructions in-between hooks to the attestation. As long as the malicious instructions do not interfere with attesting benign code, e.g., intended control flow, the attestation can be bypassed. In the following, we describe how we implement the attacks to bypass SMART and C-FLAT.

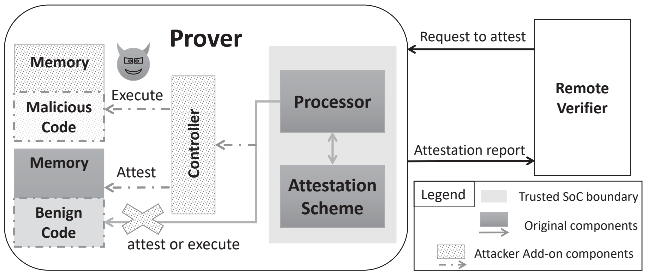


Figure 2: Memory bank attack on attestation schemes

A. SMART

SMART [9] is a static attestation scheme that establishes a root of trust in low-end embedded systems with minimal hardware components. It targets microprocessors that are able to execute code from an external memory, whereas the attestation code and key reside in an internal ROM and are protected by access control policies of a memory protection unit (MPU). When an attestation request is received, the *atomic* attestation code in ROM computes a HMAC of a region of code memory, provided in the attestation request. Then the attested code executes *atomically*.

Detecting Attestation Execution. By eavesdropping in the communication channel between the verifier and the prover for an attestation request, we determine when the attestation engine is about to run in order to launch a TOCTOU attack. Although this is permissible by the adversary model in SMART, we choose not to tackle the detection problem this way. Instead, we examine a side-channel that is inherent to the SMART design, by placing a monitor on the address bus between the processor and memory to capture which addresses are being accessed. Using the access patterns, we are able to discern whether a CPU is executing from external memory or from the internal ROM. Since SMART is prototyped on the open-source MSP430[1], it utilizes a von Neumann architecture, where data and instructions are accessed over the same address space but are structured such that they reside in different sections of memory. Hence, we can extract and filter out data accesses, leaving behind accesses to code memory. In doing so, we observe the time-frame that it takes the internal ROM to set up the attestation environment, followed by the linear scan of code addresses, then the subsequent execution of external code. On processors with modified Harvard architecture, a temporary halt in accesses to code memory would be recognized, as the ROM code starts executing. We then observe a linear scan over an address range, as code is being read and hashed by the attestation code. A break is then noticed as the ROM code cleans up memory, followed by the continued access to program memory for execution. Utilizing this, we perform one of the following attacks to mount a TOCTOU attack.

Blind Execution of Malicious Software. Since code memory remains external to the SoC, we splice the address bus, add a new memory chip containing malicious code and utilize the monitor to detect when the attestation code runs. When attesting, we bank to the memory with the intended code. When executing, we bank to the malicious code memory, allowing SMART to report valid attestation results while malicious code is actually executed by CPU during periods of no attestation.

Leakage of Secrets via Data Memory Banking. As the attestation code runs, temporary values are saved in memory, assuming SMART implementation utilizes off-chip memory to store temporary values. We use the monitor to detect when the attestation code runs. As data memory is accessed to store temporary values, we bank memories to allow for the leakage of values. We perform this by physically tampering with the address lines between the processor and the memory. As the monitor detects when SMART is about to perform its cleanup routines, we bank to a different portion of memory, leaving the ROM code to erase the wrong portion of memory. By reading the SMART secrets from memory, we are able to reconstruct the attestation secret and fake a valid response.

B. C-FLAT

C-FLAT [3] is a runtime attestation scheme that aims to measure and report the control-flow behavior of an executing code. It instruments all branch instructions such that they are intercepted by a *runtime tracer* (RTT). The RTT recovers the source and destination addresses of the branch as well as its type, which are then passed to the *measurement engine* (ME). The ME is responsible for computing a hash over the reported branches and these hash measurements are secured by running in a TrustZone secure world. In this way, a runtime control-flow attestation report is generated and verified against previously computed control-flow traces stored in a trusted verifier party.

C-FLAT is susceptible to two TOCTOU attacks assuming that the attacker has physical access to the code memory : 1) replacing instructions within a basic block with malicious ones; and 2) refactoring the control-flow graph (CFG) of an arbitrary program to match a benign CFG protected by C-FLAT. Both attacks exploit the fact that C-FLAT attests *only* control flow when exiting a basic block but not the executed instructions themselves. Hence, intermediate instructions within the basic block can be arbitrarily replaced by malicious executable code by a stronger adversary with physical access to the code

[1]http://opencores.org/project,openmsp430

memory, as long as the control flow of the code remains unchanged and the expected attestation report is not violated. These attacks are also applicable to the hardware-assisted control-flow attestation scheme LO-FAT [7] since it also only attests control flow.

We chose to implement a TOCTOU attack against one of the case studies presented in [3], namely the syringe pump program responsible for dispensing intravenous (IV) fluids. Our attack goal is to dispense liquid in incorrect volumes at unexpected times, thereby, disrupting the correct flow of IV fluids. We only demonstrate the second attack variant, however, the first variant of the attack is also easily feasible by replacing the original instructions within the basic block with malicious ones. This allows the original RTT hooks into the ME to compute a valid attestation report as it is based upon the source and destination addresses of a branch and its type.

In place of the original program that manages liquid dispensing and withdrawal, we implement a malicious program that chooses a random value to dispense by modifying the `set-quantity` function and additionally creates compound dispense and withdraw triggers for the `move-syringe` function. We embed this code in the original program, which creates new edges in the CFG of the syringe pump program. Our new edges would violate C-FLAT's attestation report for the benign syringe pump program.

To avoid triggering C-FLAT, we refactor the CFG of our attacker syringe pump program using the REpsych tool[2] to construct the desired CFG. The REpsych tool is an IDA plugin that translates a source image into a functioning program whose CFG is the image. We used the original syringe pump's CFG as a source image, and our modified syringe pump program as the target. This allowed us to generate a program with alternative functionality, but equivalent CFG to the original syringe pump program. We then recompute the attestation report using C-FLAT's tools[3]. The attacker program's attestation report matched the original syringe pump program's attestation report after CFG refactoring. Thus, we were able to accurately execute the attacker program without violating C-FLAT's protection.

IV. ATRIUM

We present ATRIUM a runtime attestation scheme targeting bare-metal embedded systems software. ATRIUM comprises a remote embedded system, called in this context the prover $\mathcal{P}rv$, and a trusted verifier $\mathcal{V}rf$. The $\mathcal{P}rv$ is deployed in-field such that the adversary has physical access to its memory. Typically, both $\mathcal{V}rf$ and $\mathcal{P}rv$ have access to the binary code of the program P to be attested on $\mathcal{P}rv$. Note that, in practice, it may not be feasible to apply runtime attestation to the entire program code due to obvious efficiency reasons, but it can be applied to pre-defined security-critical code regions.

A. Adversary Model and Assumptions

In addition to the standard capabilities of the adversary in typical remote attestation schemes, which assume software-only attacks, our adversary can also perform runtime attacks (§ II). Furthermore, we assume a stronger adversary that has physical access to the $\mathcal{P}rv$'s memory and can manipulate the program code at runtime and, therefore, is able to mount a TOCTOU attack (§ III). However, the adversary cannot modify memory reserved and used by ATRIUM itself – this memory is hardware-protected and not mapped to the software-accessible address space. *Data-oriented programming attacks* [13] that do not affect the control flow as well as invasive physical attacks on the SoC that aim at extracting secret keys are out of scope. This assumption is reasonable, since an adversary is more likely to mount a simple physical attack on the memory as we demonstrated in § III, rather than expensive sophisticated invasive attacks on the chip that can destruct it eventually.

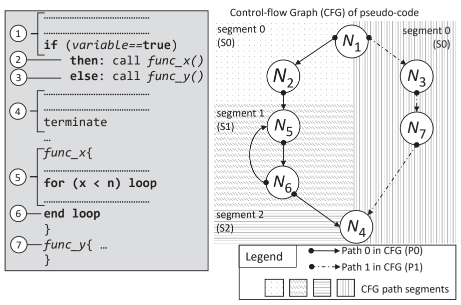
B. Runtime Attestation: High-Level Scheme

Inspired by C-FLAT [3] (described in § III-B) and the hardware-assisted scheme LO-FAT [7], ATRIUM performs attestation of an executing program code at runtime. However, unlike both schemes, it measures both the executed instructions (to detect the more advanced TOCTOU attacks described in § III) and control flow (to detect runtime attacks).

Similar to C-FLAT, our attestation mechanism relies on $\mathcal{V}rf$ performing one-time offline pre-processing to generate the CFG of program P (including expected loop execution information) by means of static and dynamic analysis. $\mathcal{V}rf$ computes cryptographic hash measurements over the instructions and addresses of basic blocks along legal CFG paths and stores them in a reference database. $\mathcal{V}rf$ initiates the attestation by sending $\mathcal{P}rv$ benign input in_b, the code region to be attested in P, and a nonce to ensure freshness of the attestation report. $\mathcal{P}rv$ executes P on the benign inputs in_b and potentially malicious inputs in_m that are not controlled by $\mathcal{V}rf$ and may lead to the corruption of the program's control flow by means of runtime attacks (§ II). ATRIUM is triggered during the execution of the code region of interest and computes a set of hash measurements over the executed paths. When execution of the code region is complete, $\mathcal{P}rv$ generates and sends to $\mathcal{V}rf$ the final *attestation report* consisting of the concatenated set of hash values $H_0\|...\|H_n$ and the number of iterations of the hash values which correspond to executed loop paths, and a signature over $H_0\|...\|H_n$ and the nonce based on $\mathcal{P}rv$'s secret key sk. To ensure authenticity of the report, sk is stored in memory accessible only by ATRIUM. Upon receiving the report, $\mathcal{V}rf$ verifies its signature using $\mathcal{P}rv$'s public key pk and checks whether the $H_0\|...\|H_n$ values match the reference hash values under input in_b. If they match, $\mathcal{V}rf$ concludes that $\mathcal{P}rv$'s execution of the attested code region was correct in terms of executed instructions and their control flow. For better understanding, we demonstrate next by an example how the hash values are computed during attestation.

Example. A CFG of an example pseudo-code is shown in Figure 3. Each numbered node in the CFG represents the corresponding numbered *basic block* of sequential instructions in the pseudo-code and the address of the first instruction of that basic block. For example, $\mathbf{N}_5$ corresponds to the first 3

[2]https://github.com/xoreaxeaxeax/REpsych
[3]https://github.com/control-flow-attestation/c-flat



```
1  ......................
   ......................
   if (variable==true)
2      then: call func_x()
3      else: call func_y()
4  ......................
   ......................
   terminate
   ...
5  func_x{
   ......................
   for (x < n) loop
   ......................
6  end loop
   }
7  func_y{ ...
   }
```

Figure 3: Example pseudo-code and its segmented CFG

instructions outlined in the pseudo-code, constituting a single basic block, and the address of the first instruction. The CFG shown in Figure 3 has 2 main paths: **P0**, in bold, consisting of nodes N_1-N_2-N_5-N_6-N_4 and **P1**, in dashed, consisting of nodes N_1-N_3-N_7-N_4. In order to avoid combinatorial explosion of legal hash values that would occur due to multiple loop iterations, the program CFG is split into segments such that hash values for loop paths are computed separately, rather than computing a single hash value over the complete executed path of the attested region. In Figure 3, due to the loop in N_5-N_6, **P0** is sectioned into 3 segments: $S0$, $S1$ and $S2$. $S0$ comprises all nodes till loop entry at N_5, where $S1$ is initialized. $S1$ ends at the loop exit node N_6, and $S2$ is initialized at N_4 and beyond until again another loop is encountered and so on.

When path **P0** is executed and attested, ATRIUM accumulates nodes (address of the first instruction and the individual instructions in each node) along each segment and computes a hash value for each segment: a hash value $H_0 = H(N_1||N_2)$ over the nodes in $S0$ of **P0**, followed by $H_1 = H(N_5||N_6)$ over the nodes in $S1$, and $H_2 = H(N_4)$ over the nodes in $S2$, resulting in the set of hash values $H_0||H_1||H_2$ representing the executed path **P0**. **P1**, on the other hand, has no loops. Therefore, when executed the whole path is measured by a single hash value $H_3 = H(N_1||N_3||N_7||N_4)$. This CFG segmentation in hash computation allows our scheme to tackle loops and nested loops efficiently, while also allowing fine-grained attestation of their execution. It requires that ATRIUM can detect and interpret loops accurately at runtime. Unlike C-FLAT, we aim to realize this without instrumentation, hence avoiding the associated performance overheads. We present next the architecture of ATRIUM and how it interfaces directly with the processor hardware to capture at runtime every executed instruction and accurately interpret control flow and infer loop entry and exit points *without instrumentation*.

V. ATRIUM: Design and Implementation

ATRIUM is a hardware-based scheme for runtime attestation that tightly integrates with a processor, as shown in Figure 4. This allows it to extract the executed instructions and their memory addresses from the execute stage of the pipeline at runtime while the program P (that needs to be attested) executes on input values in_b and in_m. ATRIUM outputs a set of hash values $H_0\|...\|H_n$ computed over the executed path

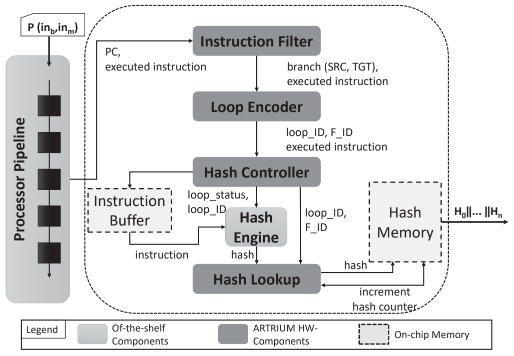


Figure 4: Architecture of ATRIUM

which get included in the attestation report. We present next the components of ATRIUM and their implementation details.

A. Instruction Filter

Upon code execution, the instruction filter extracts the current program counter (PC) and the executed instruction per clock cycle and checks whether the current instruction is a branch or jump, since such instructions reflect control-flow transitions.

Implementation. We implemented the instruction filter such that it tightly extends the execute stage of the processor from which it extracts the PC and instruction per clock cycle. If the current instruction is a control-flow instruction, its PC and the address it jumps to are stored as source–target pair, (Src, Tgt)-pair. To determine whether the branch was taken and whether control jumped forwards or backwards in memory, the PC of the next executed instruction is compared to the stored target address. Instruction filter outputs the following signals: (1) branch instructions, their type, and (Src, Tgt)-pairs and (2) basic block addresses and executed instructions.

B. Loop Encoder

As explained in § IV-B, ATRIUM handles loops and their hash computations differently. Hence, at runtime the loop encoder detects loops and identifies their entry and exit points and their depth, in case of nested loops. It checks whether the behavior of a captured branch can be inferred as returning to a loop's entry point, hence indicating a new loop iteration. The loop encoder instructs the hash controller to finalize the ongoing hash computation and initialize a new one with the entry address of a loop iteration. Furthermore, the loop encoder also detects if a branch represents a system call since system functions have to be handled specially in ATRIUM.

Implementation. To detect loops at runtime without relying on code instrumentation, we utilize a feature of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. We adopt a heuristic used in [7] to distinguish between backward branches that indicate loop entry, and branches for subroutine calls where the call target resides earlier in memory. Subroutine calls use instructions that update the *link-register* with the return address, hence, we consider any *non-linking* backwards branch as a *loop entry node*. Consequently, the basic block after the branch instruction is considered a *loop exit node*. This is based on observations of the RISC-V compiler assembly and its calling convention: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site can be compiled as a *linking* branch or inlined using the RISC-V compiler. A system call is identified by comparing its target against a predefined list of addresses of such functions and issuing a unique identifier for that function F_ID. The loop encoder stores the addresses of entry and exit nodes of each loop in a content-addressable memory (CAM) to ensure single-cycle constant-access search time. At runtime, every (Src, Tgt) is used to index the CAM to detect if a loop is re-entered or exits and to extract its loop_ID and depth (in case of nested loops). An iterations counter for each loop is maintained and updated at runtime. We detect loop exit when execution proceeds past the currently active loop exit node, either due to sequential execution or a non-linking jump, such as a *break*. The F_ID, loop_ID and loop_status signals are forwarded then to the hash controller.

C. Hash Engine and Hash Controller

The hash engine computes a hash value of each executed path within a segment (§ IV-B). The hash controller regulates the operation of the hash engine, i.e., finalizes or initiates a hash computation based on the control signals received from the loop encoder. In case the computed hash corresponds to a loop path, the hash controller sends this hash to the hash lookup and sets the search boundaries of the hash lookup to that particular current loop (necessary in case of nested loops). Otherwise, the hash value is simply stored in hash memory.

Implementation. We selected Blake2 [4] for hash computations and used the open-source hardware implementation of Blake2b, which takes as an input a message block of size 1Kbit and has a configurable digest size. We configured its digest size to 88 bits to reduce memory requirements for hash lookup and hash memory. The hash controller buffers incoming instructions from the loop encoder, aligns them in 1Kbit message blocks and feeds them to the hash engine. The hash engine requires 28 cycles to process a block, thus the hash controller issues a stall signal to the processor in case its buffer is full and the hash engine cannot digest a new message block. Therefore, system calls are handled differently because we observe that they often involve short loops that are executed arbitrarily many times, e.g., string utility functions. Hashing such a short loop path every time it executes, especially for a large number of iterations, would require the hash controller to stall the processor frequently and delibitate performance. Hence, the executed instructions along a loop path are concatenated and stored in plaintext in a dedicated CAM and sent to the hash engine only once when it is first encountered. When the same path is executed again, it is compared with the previously recorded paths in the CAM, and a corresponding counter is incremented when a match is found, without sending it to the hash engine again. The counters are concatenated with the corresponding hash values in the final attestation report.

[4]https://blake2.net/

Upon finalizing a hash computation, the hash controller checks, whether the resulting hash is computed over a path within a loop or not. If it is computed over a path loop, it forwards the resulting hash value from the hash engine synchronized with its corresponding loop_ID to the hash lookup.

D. Hash Lookup

The hash lookup is dedicated to storing and tracking hash values during loop iterations efficiently. Once a hash value is ready, the hash controller forwards it to the hash lookup, which searches within the current loop's list of hash values for a match. If not found, then the hash value is appended to the list. The hash lookup also maintains a counter per loop path which is incremented when its corresponding hash is encountered.

Implementation. To avoid multiple memory accesses due to sequential search of a particular hash value, we implement the hash lookup as a set of CAMs, whose number can be configured based on the system's requirements. A CAM is dedicated for every active loop, so the number of CAMs is determined by the maximum number of nested loops that ATRIUM is configured to track concurrently. Each CAM has a configurable capacity of (n,m) bits, where n is the maximum number of entries and m is number of bits per entry and a counter to maintain the occupied number of entries. When a loop is detected, the hash controller sends the hash lookup to reserve a CAM for it and reset its counter to zero. The CAM holds the computed hash values of a currently executing loop temporarily till the loop exits. Each time a path in the pertinent loop is executed, its computed hash value is looked up in the associated CAM. If a match is not found, i.e., this path has not been executed before, then its hash value is appended to the CAM. When a new loop is detected and all CAMs are occupied, a CAM that was reserved for a loop that already exit (and will not be executed again) is freed and re-used. If a path does not belong to a loop, then its hash value is used to update the hash memory directly.

E. Hash Memory

All computed hash values are stored in a dedicated memory. After the execution of the code region to be attested completes, these hash values are assembled and a digital signature is computed over them. The hash values $H_0\|...\|H_n$ and their signature are then transmitted to $\mathcal{V}rf$.

Implementation. An on-chip hash memory is dedicated to store all computed hash values during a single attestation run of the pertinent code region. The sequence of the storage of the hash values in memory indicates the order of the first occurrence of their corresponding code segments during execution. It is necessary to maintain this order and report $H_0\|...\|H_n$ in the same sequence to $\mathcal{V}rf$ for correctly verifying execution. In our FPGA prototyping of ATRIUM (cf. § VI), we configure the hash memory as on-chip block RAM (BRAM) of configurable capacity with each entry occupying 88 bits for hash digest and 8 bits for its counter. The capacity is configured according to our attestation requirements, i.e., the maximum number of CFG segments an attested code region would consist of. Alternatively, for constrained embedded systems, we can reduce the memory requirements by streaming the hash values (or every batch of them) as soon as they get generated to the $\mathcal{V}rf$.

VI. Evaluation & Security Considerations

A. Performance & Area Evaluation

We implemented ATRIUM in Verilog, interfaced it with the open-source RISC-V Pulpino core [5], and simulated and synthesized it. Performance and functionality were evaluated using a suite of microprocessor benchmarks including *Dhrystone*, *mt-matmul*, *rsort*, *spvm* and *towers*.

Functionality. We extended the Pulpino RTL with ATRIUM and performed cycle-accurate simulation on ModelSim while executing the aforementioned benchmarks. We confirm correct functionality of ATRIUM by comparing simulation results with reference execution profiles of the benchmarks, which we extracted by running the benchmarks on standalone Pulpino without ATRIUM and analyzing the execution trace.

Area and Memory. Area utilization depends on the configurations of the hash lookup and hash memory of ATRIUM. For our evaluation, we configured the hash lookup with 8 CAMs, each CAM with $n = 8$ entries and each entry being $m = 88$ bits. This allows ATRIUM to track up to 8 active nested loops at once with a maximum of 8 different $88 - bit$ path hashes per loop. On synthesizing ATRIUM using Xilinx Vivado on a Zedboard (Virtex-7 XC7Z020 FPGA), we show the overall area utilization to be 15% of slice registers and 20% of slice LUTs of this FPGA, while only one 18Kbit BRAM is required for the hash memory.

Performance. Implementation results indicate that ATRIUM can operate at a maximum clock frequency of 70 MHz on a Zedboard (Virtex-7 xc7z020 FPGA) and is, hence, on par with the Pulpino's maximum clock frequency of 50 MHz on the same board. Performance experiments show an overhead of 1.97% for *Dhrystone*, 12.23% for *mt-matmul*, 22.69% for *rsort*, 6.06% for *spvm* and 1.7% for *towers*. Since ATRIUM components run on par with Pulpino, performance loss is caused by the hash function, as the processor stalls occur *only* when the currently executed path has ended and needs to be hashed while the hash engine is still processing the previously executed path and is not ready for input. This overhead is incurred for loops with paths whose number of executed instructions are less than the required number of cycles for the hash engine to finalize its computation (28 cycles for the chosen hash function). To mitigate this overhead, the hash engine should be clocked at a higher frequency than the processor if possible.

B. Security Considerations.

We assume that the used cryptographic primitives are secure. Upon receiving an attestation request, $\mathcal{P}rv$ generates and sends the list of computed hash values $H_0\|...\|H_n$ along with a digital signature computed over it and a nonce provided by $\mathcal{V}rf$ and signed by $\mathcal{P}rv$'s secret key sk. The signature guarantees the authenticity of the attestation report while the nonce ensures its freshness. By verifying the signature, checking the value of the nonce, and comparing the received hashes to their expected values stored in $\mathcal{V}rf$'s database, $\mathcal{V}rf$ gains assurance of the correct execution (both instruction and their control flow) of the current program on $\mathcal{P}rv$. We consider three classes of attacks that can be mounted on ATRIUM.

Malware and Network Attacks. ATRIUM detects malicious software modification introduced by the adversary, as every executed instruction is included in the hash computation. To evade detection, finding a second image that maps to same hash value is required. However, that is infeasible since the hash engine is second pre-image resistant. Forging the signature or replaying an old signature is also not feasible, due to security of signature scheme and to the nonce being long enough.

Runtime Attacks. Since basic block addresses are included in hash computations along with the executed instructions, the hash values computed in ATRIUM reflect the control flow of the executed path. Being tightly integrated with the processor, ATRIUM is guaranteed to track and record every control-flow event executed. An attacker who knows the program code P or CFG(P) can try to bypass ATRIUM by searching for a second pre-image of the corresponding hash. However, by using cryptographically-secure hash function, finding collisions is computationally infeasible.

Physical Attacks. An adversary with physical access to $\mathcal{P}rv$ can try to manipulate the program code in $\mathcal{P}rv$'s memory at runtime, i.e, between time of attestation and time of execution. However, in ATRIUM attestation is performed *during* execution. Therefore, it is guaranteed that *every instruction* that is executed on $\mathcal{P}rv$ will be included in the hash generation, and consequently any manipulation will be detected by $\mathcal{V}rf$, as the generated hash values would not match $\mathcal{V}rf$'s expectations. This defends against TOCTOU attacks that can occur when attestation is *followed* by execution, as was the case for both SMART [9] and C-FLAT [3]. Finally, fault injection attacks which target the SoC clock and cause unintended behavior would also be detected by $\mathcal{V}rf$, as long as the attacks affect the instructions executed or their control flow. Note that, expensive invasive/semi-invasive physical attacks on the SoC are considered out of scope in this work.

VII. Related Work

Attestation Schemes. Existing static attestation schemes such as software-based [14], [20], hardware-based [21], [17], and hybrid [15], [9] attestation schemes are vulnerable to runtime attacks. Control-flow attestation (C-FLAT) aims at enhancing the security of static attestation schemes by additionally hashing the code's execution control flow. This enables the detection of code-reuse and non-control data attacks that divert the execution flow. However, due to frequent hash calculations and context switching (on TrustZone), C-FLAT incurs high performance overhead. LO-FAT [7] leverages hardware assistance to track and measure control flow, thus, overcoming the limitations of C-FLAT and enabling efficient attestation of *uninstrumented* code. LO-FAT, however, incurs significant area overhead due to its on-chip memory requirements (up to 49 36Kbit Block RAMs are used sparsely to store counters of

[5]https://github.com/pulp-platform/pulpino

loops' paths). Finally, in a stronger adversary model with physical access to the prover's device, these schemes are vulnerable to Time of Check Time of Use (TOCTOU) attacks. ATRIUM mitigates this by providing both static and control-flow attestation in a stronger (and more realistic) adversary model efficiently.

Authenticated Memory Modules. Authenticated Memory Modules (such as Intel Authenticated Flash [1]) aim at resisting physical attacks on external memory by preserving the memory's integrity. However, they are insecure under an adversary model with physical access. Moreover, they do not authenticate the control flow of the execution. On the contrary, ATRIUM provides an additional defense against software runtime attacks by coupling the attestation of both the instructions and their control flow with their execution to eliminate any room for TOCTOU attacks.

Memory Authentication. Such schemes [8], [6] aim at resisting physical attacks on external memory. However, they incur high performance overhead by authenticating memory blocks before execution and are susceptible to runtime attacks. ATRIUM detects both runtime attacks and physical attacks on code memory while incurring minimal overhead.

Hardware Security Architectures. Finally, hardware security architectures (such as Intel SGX) provide memory authentication as well as static attestation. However, such architectures are not designed to target low-end embedded devices. Furthermore, they only provide static attestation and therefore cannot meet the goals that we target. Nevertheless, they provide security features complementary to our work.

VIII. Conclusion

Due to the ubiquity of interconnected embedded systems, software running on these devices have become vulnerable to remote software attacks. Previous attestation schemes have been proposed to detect these attacks while always ruling out physical attacks. In this paper, we showed that physical attacks on the system's code memory are indeed feasible. We presented a hardware-based efficient scheme ATRIUM that allows precise attestation of both executed instructions as well as their control flow. ATRIUM is the first attestation scheme to provide security guarantees against a stronger adversary with physical access to code memory, and does not require any code instrumentation (compliant to legacy software) or instruction set extension. Our proof-of-concept implementation is highly efficient with reasonable performance impact on the attested software at an expense of minimal area overhead and memory.

Acknowledgments. We thank the authors of SMART for sharing their scheme. This work was supported by the German Science Foundation CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant No. 643964 (SUPERCLOUD), the U.S. Department of Energy through the Early Career Award (DE-SC0016180), the National Science Foundation Graduate Research Fellowship Program under grant No. 1144246, and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

References

[1] Intel Authenticated Flash. www.design-reuse.com/articles/16975.
[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity: Principles, implementations, and applications". *ACM Transactions on Information and System Security*, 2009.
[3] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. "C-FLAT: Control-flow attestation for embedded systems software". In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.
[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. "When good instructions go bad: Generalizing return-oriented programming to RISC". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.
[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. "Non-control-data attacks are realistic threats". In *USENIX Security Symposium*, 2005.
[6] R. de Clercq, R. De Keulenaer, P. Maena, B. Preneel, B. De Sutter, and I. Verbauwhede. "SCM: Secure code memory architecture". In *ACM Symposium on Information, Computer and Communications Security*. ACM, 2017.
[7] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. "LO-FAT: Low-overhead control flow attestation in hardware". In *Design Automation Conference*, 2017.
[8] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. "Hardware mechanisms for memory authentication: A survey of existing techniques and engines". In *Transactions on Computational Science IV*. 2009.
[9] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. "SMART: Secure and minimal architecture for (establishing dynamic) root of trust". In *Annual Network and Distributed System Security Symposium*, 2012.
[10] A. Francillon and C. Castelluccia. "Code injection attacks on Harvard-architecture devices". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.
[11] V. Haldar, D. Chandra, and M. Franz. "Semantic remote attestation: A virtual machine directed approach to trusted computing". In *Virtual Machine Research And Technology Symposium*, 2004.
[12] Hewlett-Packard. "Data execution prevention", 2006.
[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-oriented programming: On the effectiveness of non-control data attacks". In *IEEE Symposium on Security and Privacy*, 2016.
[14] R. Kennell and L. H. Jamieson. "Establishing the genuinity of remote computer systems". In *USENIX Security Symposium*, 2003.
[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. "TrustLite: A security architecture for tiny embedded devices". In *ACM SIGOPS EuroSys*, 2014.
[16] T. Kornau. "Return oriented programming for the ARM architecture". Master's thesis, Ruhr-University Bochum, 2009.
[17] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. "New results for timing-based attestation". In *IEEE Symposium on Security and Privacy*, 2012.
[18] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. "Code-pointer integrity". In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.
[19] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. "SoK: Automated software diversity". In *IEEE Symposium on Security and Privacy*, 2014.
[20] Y. Li, J. M. McCune, and A. Perrig. "VIPER: Verifying the integrity of peripherals' firmware". In *ACM SIGSAC Conference on Computer and Communications Security*, 2011.
[21] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. "Copilot – A coprocessor-based kernel runtime integrity monitor". In *USENIX Security Symposium*, 2004.
[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. "Design and implementation of a tcg-based integrity measurement architecture". In *USENIX Security Symposium*, 2004.
[23] H. Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In *ACM SIGSAC Conference on Computer and Communications Security*, 2007.
[24] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal war in memory". In *IEEE Symposium on Security and Privacy*, 2013.