LO-FAT: Low-Overhead Control Flow ATtestation in Hardware

Ghada Dessouky[1], Shaza Zeitouni[1], Thomas Nyman[2,3], Andrew Paverd[2], Lucas Davi[4], Patrick Koeberl[5], N. Asokan[2], Ahmad-Reza Sadeghi[1]

[1]Technische Universität Darmstadt, Germany
{ghada.dessouky,shaza.zeitouni,ahmad.sadeghi}@trust.tu-darmstadt.de

[2]Aalto University, Finland
thomas.nyman@aalto.fi, andrew.paverd@ieee.org, asokan@acm.org

[3]Trustonic, Finland
thomas.nyman@trustonic.com

[4]University of Duisburg-Essen, Germany
lucas.davi@uni-due.de

[5]Intel Labs, Germany
patrick.koeberl@intel.com

ABSTRACT

Attacks targeting software on embedded systems are becoming increasingly prevalent. Remote attestation is a mechanism that allows establishing trust in embedded devices. However, existing attestation schemes are either static and cannot detect control-flow attacks, or require instrumentation of software incurring high performance overheads. To overcome these limitations, we present LO-FAT, the first *practical hardware-based* approach to control-flow attestation. By leveraging existing processor hardware features and commonly-used IP blocks, our approach enables efficient control-flow attestation without requiring software instrumentation. We show that our proof-of-concept implementation based on a RISC-V SoC incurs no processor stalls and requires reasonable area overhead.

1 Introduction

Embedded systems have been facing a variety of security challenges for decades [25] which are becoming increasingly prevalent with emerging trends such as collaborative Internet of Things (IoT). A recent prominent example is *Mirai* malware[1] in October 2016, where a series of Distributed Denial-of-Service (DDoS) attacks against the DNS system disrupted a number of prominent websites.These attacks were perpetrated by IoT devices, including routers, DVRs, and web-enabled security cameras, that had been compromised by the Mirai malware.

Increasingly, attacks against embedded systems aim to exploit software vulnerabilities. In 2015, a remotely exploitable buffer overflow vulnerability was found in the *USB over IP* software used in millions of residential gateways and wireless routers supplied by prominent manufacturers[2]. In 2014, a memory corruption flaw was found in the embedded webserver software used by over 200 different models of embedded devices, affecting at least 12 million devices, many of which still remain vulnerable today[3].

Remote attestation is an important class of security mechanisms designed to detect software attacks. In principle, remote attestation allows one entity (the *verifier*) to ascertain the precise state of the software running on a remote system (the *prover*). However, most attestation schemes are *static* in that they attest the software initially loaded by the prover before it begins executing. Although useful, this still leaves the system vulnerable to *run-time* software attacks. If the adversary gains control of the stack or heap, (s)he can alter control-flow information to subvert the control flow of the target program, and mount a *code-reuse attack*. Similarly, in *non-control data* attacks [8], the adversary modifies strategic data variables to cause a permissible but unintended control flow change (e.g., executing a privileged instruction sequence). Traditionally, code-reuse attacks are mitigated using techniques such as control-flow integrity (CFI) [1]. However, CFI cannot prevent non-control data attacks, since these do not violate control-flow integrity. Neither of these types of attacks can be detected by static attestation.

To overcome these challenges *control-flow attestation* [2] was proposed very recently, enabling the prover to precisely report the control flow of application software to the verifier while giving assurance on control-flow integrity and detection of non-control data attacks. The attestation mechanism of [2] requires an isolated execution environment (e.g., ARM TrustZone, Intel SGX) to protect it against potentially compromised application software. However, implementing control-flow attestation in software has two limitations: Firstly, in order to detect control-flow events, the application software must be *instrumented* prior to deployment. Non-instrumented or incorrectly-instrumented software cannot be attested. The instrumentation rewrites all control-flow instructions (e.g., `branch`, `return`, etc.) in order to transfer control to the attestation software. Secondly, the attestation software runs on the main processor which incurs significant performance penalties because single control-flow instructions are essentially replaced with relatively many numbers of instructions in order to track and record the control-flow event (e.g., update a running hash value). As we elaborate in §7, some existing hardware approaches, such as debugging and tracing features in modern processors [14, 24] or hardware security architectures [3, 6, 9], can be used to record control flow information. However, due to the overhead they incur or the type

[1]https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html
[2]http://blog.sec-consult.com/2015/05/kcodes-netusb-how-small-taiwanese.html
[3]http://mis.fortunecook.ie/

DAC '17, June 18 - 22, 2017, Austin, TX, USA

ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062276

of information they record, these approaches are not well-suited for control-flow *attestation*.

Goals and Contributions. To overcome the limitations of a software solution, we introduce a practical hardware-based Low-Overhead Control Flow ATtestation architecture, LO-FAT. Unlike software implementations, LO-FAT can handle *unmodified application software* without instrumentation, meaning that it is transparent to legacy software. By recording the control flow in hardware in parallel to the main processor, LO-FAT does not stall the application software, thus eliminating the performance overhead of attestation in software. LO-FAT leverages existing processor features and commonly-used IP blocks and can feasibly be implemented on typical embedded systems hardware platforms.

The main contributions of this paper are:

- Design of LO-FAT, a hardware-based scheme for control-flow attestation, providing the *same security guarantees* as previous software schemes, without the performance overhead or the need for software instrumentation (§4).
- An integrated optimization for eliminating redundant attestation computation (e.g., avoiding duplication when attesting loops) and reducing the burden on the verifier (§4).
- A proof-of-concept implementation of LO-FAT on the new open-source RISC-V architecture targeting the Pulpino core for single-threaded embedded system software (§5).
- A systematic evaluation of LO-FAT in terms of the required hardware area and performance benefits (§6).

2 Problem Setting and Challenges

Remote attestation provides a well-known mechanism for detecting malware on a device. However, existing conventional (binary) attestation cannot detect run-time exploitation techniques, since run-time attacks do not not modify the program binary. Such attacks aim to subvert the intended control flow of the targeted program while it is executing. An overview of different classes of such attacks is shown in Figure 1. In general, a program reserves dedicated memories for data and code. The former is marked as readable and writable *(rw)*, whereas the latter is as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten. Furthermore, any program can be abstracted through its corresponding control-flow graph (CFG) that encapsulates the valid paths a program should follow at run-time.

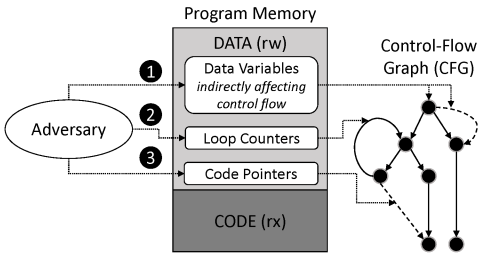

Figure 1: Overview on run-time attack classes

We can distinguish three classes of run-time attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop counter variables, and ❸ code-pointer overwrites. The most prominent run-time attacks exploit code-pointer overwrites, i.e., corruption of return addresses and function pointers. For instance, code-reuse attacks such as *Return-oriented Programming* (ROP) [23] exploit memory corruption vulnerabilities (e.g., buffer overflows) in the program and then stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the vulnerable program memory. This is exemplified by a malicious CFG edge (see dashed line for code-pointer overwrite in Figure 1). These attacks have been shown

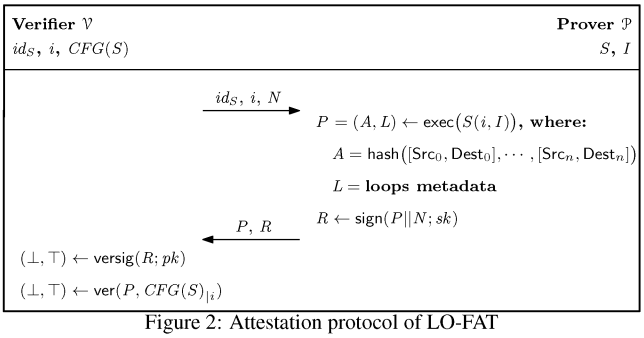

Figure 2: Attestation protocol of LO-FAT

to be a realistic threat on many processor architectures, such as Intel x86 [23], ARM [17] and embedded systems building on Atmel AVR [12]. Although countermeasures against this class of attacks exist, e.g., control-flow Integrity (CFI) [1] and code-pointer integrity (CPI) [16], they do not prevent attacks ❶ and ❷. The so-called *non-control data attacks* [8] do not compromise the control flow of a program, but cause unexpected malicious control-flow paths by corrupting data variables. In ❶, the attacker compromises data variables that are used for security decisions during program execution, e.g., corrupting an authentication variable to execute a privileged but existing path. Attack class ❷ is even more subtle as it only affects the number of times a program loop is executed. This can have severe consequences in the context of embedded system software, e.g., a syringe pump dispenses more liquid than requested (see [2]).

Control-flow attestation can cover these cases by assuring the verifier of the precise run-time control flow of the program on the embedded device. In [2], the first control-flow attestation scheme was proposed and implemented. However, it suffers from practical limitations, such as high performance overhead and the need for tedious software instrumentation.

Our work tackles the challenge of detecting attack classes ❶-❸, while addressing the limitations of recently proposed software-based control-flow attestation [2] by presenting LO-FAT, an efficient hardware-only solution.

3 System Model

Figure 2 depicts the attestation protocol of LO-FAT: the verifier $\mathcal{V}$ aims to attest the run-time control-flow (execution path) of the Program S on a remote embedded system – the prover $\mathcal{P}$. We assume that both $\mathcal{V}$ and $\mathcal{P}$ have access to the program S in binary form and that conventional static (binary) attestation assures $\mathcal{P}$ is executing the correct and unmodified program S.

First, $\mathcal{V}$ performs a one-time offline pre-processing step to generate the CFG of S (including expected loop execution information) by means of static or dynamic analysis. Next, $\mathcal{V}$ initiates the protocol by sending $\mathcal{P}$ the program input i for the program ID id_S, and the nonce N to ensure freshness of the attestation response. $\mathcal{P}$ executes S with verifier input i and a set of malicious adversary inputs I. In fact, the untrusted inputs received may corrupt the control-flow by means of the attack techniques described in §2. While S executes, LO-FAT captures the control-flow transitions and generates a cumulative authenticator A of the control-flow path taking source and destination address $(Src, Dest)$ of each branch as input. Naively storing and transmitting every single executed instruction to $\mathcal{V}$ would incur impractical memory, power and communication overheads, especially for resource-constrained embedded devices. Hence, LO-FAT follows the idea outlined in [2] and computes a cumulative cryptographic hash of the executed path. In addition, it also produces auxiliary metadata L to track program loop paths and their number of iterations (including recursive functions) thereby covering attacks of class ❷ in Figure 1. Together A and L form

a unique program path P. Lastly, upon program exit, $\mathcal{P}$ generates the *attestation report* $R = sign(P||N; sk)$, under the signing key sk, which is stored by $\mathcal{P}$ in hardware-protected secure memory, e.g., a register that is accessible only to LO-FAT. Upon receiving R, $\mathcal{V}$ verifies the signature using the verification key pk. Next, $\mathcal{V}$ checks whether the reported path P resembles a valid path in CFG under input i. If true, $\mathcal{V}$ is assured of $\mathcal{P}$'s execution.

Adversary Model and Assumptions. We assume a strong adversary that has full control over the *data memory* of $\mathcal{P}$ and can utilize standard memory corruption vulnerabilities to modify arbitrary writable memory locations. However, the adversary cannot modify program code at run-time (marked as *rx*) and cannot modify memory used by LO-FAT itself (due to hardware protection). Note that similar to all attestation schemes we consider software-only attacks and hence physical attacks on $\mathcal{P}$'s device are out of scope in this work. Also note that our scheme can detect attacks that affect the program's control-flow, but not pure data-driven attacks (that do not affect any control-flow) such as *data-oriented programming* attacks, which remain an open research problem [13].

4 LO-FAT Design

Figure 3 illustrates our architecture for LO-FAT and how it interfaces with the processor pipeline. The proposed scheme exploits branch tracking functionality inherent in any processor pipeline and re-usable IP cores such as the hash engine. We extend these with additional logic to achieve efficient tracing of control-flow information. The main LO-FAT components are the branch filter and the loop monitor. The former extracts branch instructions from the processor as it executes the attested code segment while the latter monitors program loops.

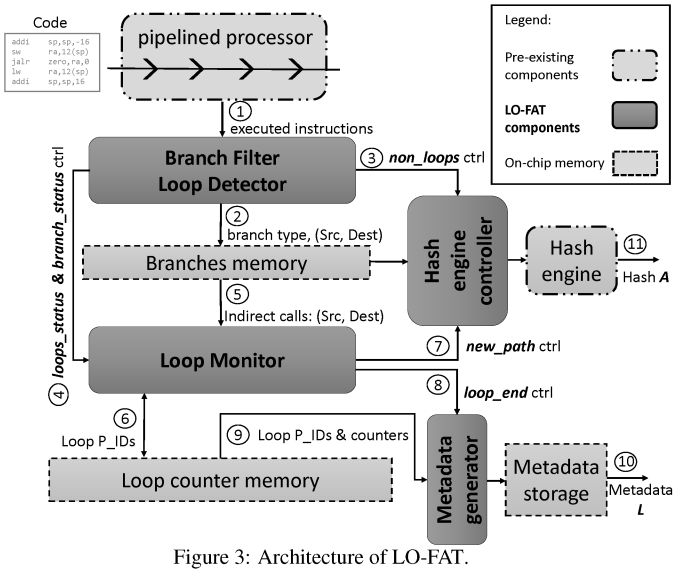

Figure 3: Architecture of LO-FAT.

Branch Filter. Upon code execution, the branch filter, which is tightly coupled to the processor, extracts the current program counter and instruction executed per clock cycle. Then it filters in every `branch`, `jump` and `return` instruction since these are the relevant instructions for control-flow attestation. The branch filter outputs a concise representation of every executed branch instruction with its source and destination address pair $(Src, Dest)$ into a dedicated branches memory and detects whether the intercepted branch is within a program loop. If not, the branch filter enables hashing of $(Src, Dest)$. Branches inside a program loop require special treatment in LO-FAT, because (i) loop counter manipulation may compromise the program's control-flow in a malicious way (§2), and (ii) naively hashing each loop iteration and path leads to a combinatorial explosion of valid hash values [2]. As such, we design LO-FAT to compress control-flow information associated with loops efficiently. As mentioned earlier in §3, we report each loop path and its number of iterations as auxiliary metadata L. However, doing so in hardware is challenging, i.e., in contrast to the most related work C-FLAT, since we do not use code instrumentation to preserve legacy compliance. Hence, the branch filter must detect and identify loop entry and exit points and their depth at run-time without instrumentation aid. We describe in §5.1 how we tackled this challenge.

Loop Monitor. When a loop is encountered, the branch filter forwards the loop entry and exit to the loop monitor. The loop monitor identifies and tracks program loops (including nested loops). When a branch inside a program loop is encountered, the branch filter forwards this information to the loop monitor which in turn encodes each path inside the loop uniquely. Simultaneously, $(Src, Dest)$ of each `branch` remains stored in the branches memory.

Another major challenge concerning loops is the hash computation and attestation overhead incurred by hashing each loop iteration. In LO-FAT, we significantly reduce the hash computation cost by only hashing each loop path once and keeping an iteration counter for each unique loop path. To achieve this, LO-FAT generates a unique path encoding for each loop path and associates an on-chip loop counter with it. The loop monitor indicates newly observed loop paths to the hash engine controller in order to hash its corresponding $(Src, Dest)$ from the branches memory. On the other hand, once the same loop path executes, LO-FAT only needs to increment the counter, i.e., not requiring further hash operations.

Upon loop exit, the loop monitor requests the metadata generator to assemble the loop auxiliary metadata based on the loops memory which contains the unique loop path encodings, their number of iterations, and indirect branch targets. This information is stored on-chip and is appended to the final hash value A computed at the end of the attested execution. Finally, a digital signature R is computed over the hash value A, metadata L and nonce N and sent to $\mathcal{V}$ for attestation (as per our protocol outlined in §3).

5 Implementation

5.1 Loop Handling

Detecting loops. As shown in Figure 3, the branch filter unit traces the instruction (and its address) executed per clock cycle and filters in ① every `branch`, `jump` and `return` instruction. It outputs a concise representation of every executed branch instruction with its $(Src, Dest)$-pair into a dedicated branch buffer (②). To compress the control-flow trace for loops, the branch filter has to detect loops. If the intercepted branch is not in a loop, the branch filter sends the control signal *non_loops_ctrl* to the existing hash engine controller to compute a hash over $(Src, Dest)$ in ③. Otherwise, the branch filter forwards the loop status (entry and exit) to the loop monitor and its depth (in case of nested loops) via the *loops_status_ctrl* signals (④).

To enable efficient run-time loop detection, we utilize a property of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. LO-FAT uses a simple heuristic to differentiate between backward branches that constitute loops, and branches for subroutine calls where the call target resides earlier in memory. Since subroutine calls use instructions that update the *link-register*, we consider the target of each *non-linking* backwards branch as a *loop entry node*. The basic block proceeding the branch instruction is considered a *loop exit node*. We base our heuristic on our observations of the RISC-V compiler assembly and the calling

convention described in the instruction manual: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site are still compiled as a *linking* branch or are optimized by traditional inlining using the RISC-V compiler.

The addresses of the entry and exit nodes of each loop are stored in registers by the loop detector and used to detect and track loop iterations and loop depth at run-time when executing nested loops. The number of loop iterations is determined by recording the number of times the loop entry node is entered within the loop. Loop termination is detected by tracking if execution proceeds to or past the currently active loop exit node, either as the result of sequential execution (e.g. in the case of a conditional branch) or a non-linking branch (e.g. break). Loop execution status is forwarded using the *loops_status_ctrl* signals to the loop monitor, as shown in Figure 3.

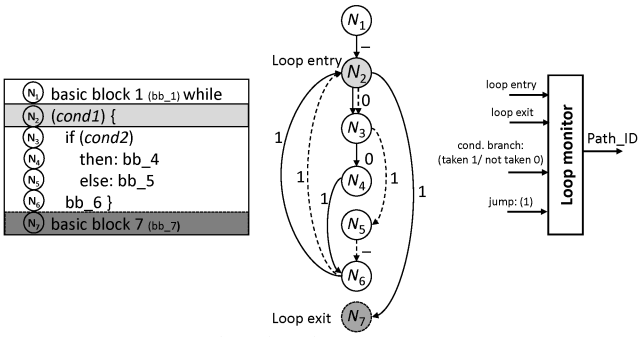

Figure 4: CFG for pseudo-code and its layout of instructions in memory.

Tracking loops. As shown in Figure 3, the loop monitor receives *branch_status_ctrl* signals from the branch filter to describe the type of intercepted branch instruction and its $(Src, Dest)$ (⑤). This branch tracking mechanism allows the loop path encoder to uniquely encode paths as they occur. Simultaneously, $(Src, Dest)$ of each `branch` along the executing loop path remain stored in the branches memory.

Figure 4 shows a sample pseudo-code and its CFG according to how the instructions would be laid out in code memory to illustrate how the loop monitor encodes the loop paths. The example code shows a *while-loop* with an *if-else* statement inside. Each basic block in the pseudo-code is represented by a node in the CFG and numbered accordingly, with loop entry and exit nodes also indicated. Within this simple loop, there are only 2 valid paths: bold path $N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6 \rightarrow N_2$ and dashed path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6 \rightarrow N_2$.

For every `conditional branch`, the processor evaluates the condition and either jumps to the computed target address (branch is taken), or continues sequentially to the next instruction address in memory (branch is not taken). Processors commonly track this branching behavior in the pipeline and may encode a taken/not-taken branch with '1'/'0'. This branch information is extracted from the processor by the branch filter and used by the loop monitor to uniquely identify and encode paths within each loop with a unique path_ID, as shown in Figure 4. In Figure 4, the dashed path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6 \rightarrow N_2$ is encoded as '011' and bold path $N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6 \rightarrow N_2$ as '0011'. Other path encodings are considered invalid and detected by the $\mathcal{V}$.

Once a loop path is completed, this unique path_ID is used to index *loop counter* memory, in which the number of iterations for each corresponding path is saved (⑥) in Figure 3. A counter value of zero indicates the first time a particular path is executed. This is forwarded by the loop monitor into the hash engine controller using *new_path_ctrl* signals (⑦) to enable hashing of corresponding $(Src, Dest)$ pairs. Otherwise, the counter is simply incremented.

To ensure constant-time, single-cycle memory access latency, we implement *loop counter memory* as on-chip memory indexed by the unique loop path encodings. However, this consumes a dedicated sparsely-utilized memory which is often a constrained resource on low-end embedded devices. In light of this, LO-FAT allows configuring the granularity of the control-flow tracking according to the availability of memory resources.

Once a loop exits, this is identified by the loop monitor and indicated in the *loop_end_ctrl* signals sent to the metadata generator (⑧). The metadata generator assembles the loop auxiliary metadata from the loops memory - this consists of the unique loop path encodings in order of first occurrence, the number of iterations of each path, and the indirect branch targets encountered in this loop (⑨). This fine-grained auxiliary information on loop execution is stored on-chip (⑩) and is appended to the final hash value computed at the end of the attested execution (⑪). Finally, a digital signature is computed over the hash value, metadata and nonce N, and sent to $\mathcal{V}$ for attestation. Handling indirect branches in loops is yet another implementation challenge we discuss next.

5.2 Handling Indirect Branches in Loops

Indirect branches can involve any arbitrary number of targets which can never be exhaustively identified using static analysis. To uniquely identify loop paths with indirect branches (calls and returns), we would need to include the 32-bit target addresses into the path encodings, which would require infeasibly high memory requirements for loop path-indexed memory. Instead, we re-encode the addresses using a smaller number of n bits, allowing a maximum number of 2^n-1 possible targets for each loop. Target addresses are encoded at run-time and stored in a register file, which is implemented as 2 interleaved CAMs to ensure low-latency constant-time access. When a target address is encountered that exceeds the configured limit, we report this in the encoding to the $\mathcal{V}$ by an all-zero code. LO-FAT is designed such that the maximum number of branches per loop path and the maximum number of possible target addresses (of indirect branches) to track is configurable in a trade-off between granularity and availability of on-chip memory. Tracking ℓ branches per path in a loop requires $8 \times 2^\ell$ bits memory. In our implementation, we configure $n = 4$ to track up to 16 possible indirect branch targets for a given loop and $\ell = 16$ such that LO-FAT can handle a maximum of 16 branches per loop path (every additional indirect branch tracked reduces the maximum number of possible conditional branches by n) and depth of up to 3 nested loops, which requires a dedicated 1.5 Mbits memory that is synthesized as block RAM (BRAM) when prototyping on FPGA. Once a loop exists, its memory is re-used for other subsequent loop executions.

Loop metadata. The measurement in A is a single hash computation of $(Src, Dest)$ pairs of executed loop paths. To enable $\mathcal{V}$ to reconstruct the final hash value, metadata L of the loops serves as helper data and provides $\mathcal{V}$ with fine-grained insight into the execution of the loops. L contains the encodings of executed paths in each loop, the order of first occurrence of each executed path, and number of iterations per loop path and indirect branch targets.

5.3 Hash Engine

A single hash measurement A is computed on the full execution path, along with auxiliary loop metadata L. We employ a SHA-3 512-bit open-source engine[4] operating at a maximum clock frequency of 150 MHz. It consists of a permutation module which operates on a message block size of 576-bit. User input is absorbed by the core first into a padding module to assemble the 576-bit block size. Once this padding is full, the permutation module begins computation on

[4] http://opencores.org/project,sha3

input. In LO-FAT, the engine can absorb a 64-bit input $(Src, Dest)$-pair every clock cycle into the padding module for 9 clock cycles, after which the 576-bit buffer becomes full and notifies the permutation module to begin its computation. Once full, the padding buffer cannot absorb further input for 3 clock cycles after which it resumes normally. Therefore, a small cache buffer is configured at the hash engine input to prevent dropping of $(Src, Dest)$-pairs if they arrive during these cycles where the padding buffer is full. Using this hash engine, an unlimited message size can be hashed while indicating the end of streaming $(Src, Dest)$-pairs when the execution of attested software is completed.

6 Evaluation

We present a proof-of-concept implementation of LO-FAT on Pulpino [18], the first open-source RISC-V-based microcontroller SoC [19]. It is based on a single 32-bit 4-stage minimal RISC-V core targeting low-end embedded systems. We augment the RISC-V processor pipeline to interface with the LO-FAT branch filter to extract control-flow signals required for execution flow tracing. LO-FAT can be easily integrated into any low-end embedded processor as it does not require modifications to the ISA.

6.1 Functionality and Performance

We integrated LO-FAT with Pulpino and performed cycle-accurate functional simulation of their RTL Verilog source code on ModelSim while Pulpino executed extracted code segments from real embedded applications, such as Open Syringe Pump[5], an open-source open-hardware syringe pump design. Simulation results confirmed the functionality of LO-FAT in correctly capturing and compressing the control flow (branches, loops, and nested loops) of an uninstrumented application. Since LO-FAT extracts and filters control-flow events in parallel with the processor, it does not incur any performance overhead for the attested software, as opposed to C-FLAT which incurs attestation overhead that is linearly dependent on the number of control-flow events. LO-FAT internally incurs latency of 2 clock cycles for branch instructions and loop status tracking and 5 clock cycles at loop exit for completing path_ID generation and *loop counter* memory access and update. However, LO-FAT simultaneously continues to absorb and process any incoming $(Src, Dest)$-pairs to prevent the processor from stalling or dropping trace information. Synthesis results using Xilinx Vivado indicate LO-FAT can operate at maximum clock frequency of 80 MHz on a Virtex-7 XC7Z020 FPGA device on a Zedboard. The LO-FAT units are engineered such that they operate on par with Pulpino's clock frequency, while also allowing single-cycle constant-time memory accesses for indirect branches and loops management. Eliminating the CAM access results in a much higher clock frequency if desired.

The length of the auxiliary metadata (L) that must be sent to $\mathcal{V}$ depends on the number of loops executed, the number of different paths per loop, and the number of indirect branch targets encountered in the attested code.

6.2 Area

On a Virtex-7 XC7Z020, LO-FAT consumes 4% of the available registers and 6% of available LUTs, which amounts to an average of 20%additional logic overhead to the Pulpino SoC. 49 36Kbit Block RAM (BRAMs) are utilized, most of which are dedicated for the sparse loop path-indexed memories to ensure constant-time single-cycle access. Therefore, its width depends on the configured maximum number of indirect branches allowed in each loop path and number of bits required to encode them, as discussed in §5.2. In our implementation, the loop monitor is configured to tackle up to 4 indirect branches and requires 10 bits to encode them in $Path\_ID$, resulting in 16 BRAMs per loop. Since we allow up to 3 levels of nested loops, we require 48 BRAMs. Configuring these parameters to lower numbers or leveraging CAMs instead reduces the memory requirements significantly at the expense of coarser granularity or additional logic overhead respectively.

[5] https://hackaday.io/project/1838-open-syringe-pump

6.3 Security

The primary security requirement of LO-FAT is to provide an *accurate*, *complete*, *authentic*, and *fresh* attestation of $\mathcal{P}$'s control flow. This requires an integrity-protected mechanism for recording control-flow information and unforgeably communicating this to $\mathcal{V}$.

Control-Flow Recording. One of the main contributions of LO-FAT is using low-overhead hardware extensions to record control-flow information preventing it from being modified or subverted by malicious software. The on-chip memory employed by LO-FAT for storing the $(Src, Dest)$ addresses prior to their hashing is also assumed to be protected from adversarial access. The hardware extensions are guaranteed to receive every control-flow event from the processor, thus ensuring that the complete control flow is recorded. All $(Src, Dest)$ addresses are cryptographically hashed resulting in the authenticator A. The auxiliary metadata L records (1) the unique paths within each loop; (2) the number of repetitions of each path; and (3) all indirect branches encountered within loops.

Attestation Protocol. LO-FAT makes use of the widely-used secure challenge-response attestation protocol. As explained in §3, $\mathcal{P}$ sends the recorded program path P along with a digital signature over P and a nonce supplied by $\mathcal{V}$. If $\mathcal{P}$'s signing key has not been compromised, this signature guarantees the authenticity of the attestation, and the inclusion of the challenge nonce ensures freshness. Our assumed software adversary cannot compromise the signing key because it is stored in hardware-protected secure memory. Any tampering with the attestation messages can be detected by $\mathcal{V}$.

Given that the control flow recording and the signing key is protected from software attacks, the resulting attestation report provided by LO-FAT is accurate, complete, authentic, and fresh. Since $\mathcal{P}$'s code is immutable and is statically attested at boot time, $\mathcal{V}$ has complete information about $\mathcal{P}$'s execution. As described in §3, $\mathcal{V}$ also has access to the CFG of the attested software, which it can use to identify permissible control flows and detect control-flow attacks or non-control data attacks.

7 Related Work

Remote Attestation. Most prior work focuses on *static* remote attestation [7, 11, 21], which is orthogonal to run-time attestation – the focus of this paper. Software-based attestation [22] can, under strict assumptions, enable static attestation of legacy devices without hardware-based trust anchors. Property-based attestation [20] can attest behavioral characteristics of a program, with the assistance of a trusted third-party. However, none of these can attest control-flow at machine code instruction level.

Prior work on run-time attestation focuses on specific aspects of a program's execution. ReDAS [15] attests program data invariants, such as the integrity of a function's base pointer, at each system call. Trusted virtual containers [4] attest the run-time launch order of application modules – a form of coarse-grained control-flow attestation that does not include internal control flows within modules. DynIMA [10] uses dynamic taint analysis and tracing to attest run-time properties that may be symptomatic of run-time attacks. However, it does not cover non-control data attacks and incurs high performance overhead due to dynamic taint analysis.

C-FLAT [2] is a fine-grained control-flow attestation scheme. LO-FAT also leverages the idea of attesting the control flow of an application by computing a cumulative hash of executed branches but with several fundamental differences. C-FLAT requires *instrumentation* of all control-flow instructions thereby violating legacy compliance. In contrast, LO-FAT does not require any binary rewriting. C-FLAT requires complete coverage in the offline binary analysis, as un-instrumented control-flow instructions could be exploited to mount undetectable attacks. This is not possible in LO-FAT as every executed branch is monitored by design. Finally, C-FLAT incurs significant performance overhead, whereas LO-FAT incurs no performance overhead due to its efficient hardware support for control-flow attestation.

Tracing and Debug Mechanisms. Intel processors provide the *Last Branch Record* (LBR) and *Branch Trace Store* (BTS) mechanisms, which can be used to trace control-flow events [24]. However, the overhead incurred by these debugging mechanisms makes them unsuitable for control-flow attestation. Recently, Intel processors introduced *Intel Processor Trace* (IPT) [14], a low-overhead execution tracing feature that collects more tracing information than BTS (including execution mode and timing information). However, IPT cannot be directly used for control-flow attestation as it only reports control-flow events that cannot be inferred from static analysis. ARM's CoreSight[6] debug and trace architecture provides a mechanism to access trace information from different hardware trace components. However, high-throughput tracing on ARM typically requires the use of proprietary hardware.

Hardware-Assisted Security. Recent work [5, 26] developed a generic architecture for enforcing a diverse range of SoC security policies. Each IP block has an individually-customized security wrapper that sends security-relevant events and information to a central security controller to enforce individual security policies for each IP. However, this incurs high memory and logic complexity overhead as the number of IPs increases. It has further been proposed [3, 6] that this could be made more practical by re-purposing design-for-debug features found on many SoCs – a promising approach which could complement LO-FAT in future.

Sofia [9] is a recent hardware-assisted architecture for enforcing control-flow integrity (CFI). It encrypts instructions with CFI-dependent data, such that they can only be decrypted at run-time as part of a valid control-flow path, and it ensures instruction integrity by checking MACs on groups of instructions at run-time. However, unlike LO-FAT, this requires software instrumentation and places decryption in the critical execution path, thus incurring total execution time overheads of up to 110%.

8 Conclusion

Due to the increasing prevalence of interconnected embedded systems, software running on these devices have become a prime target for remote attacks. We presented in this paper the first hardware-based control-flow attestation scheme that allows precise detection of remote memory corruption attacks in embedded system software. Our architecture, LO-FAT, monitors, measures and reports the program's behavior by interfacing with the processor to intercept control-flow events. LO-FAT does not require any code instrumentation (compliant to legacy software), compiler toolchain or instruction set extension. Our proof-of-concept implementation on the open-source RISC-V core is highly efficient with no performance impact on the attested software at the expense of minimal logic overhead and on-chip memory.

Acknowledgments. This work was supported by the German Science Foundation CRC 1119 CROSSING project, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant number 643964 (SUPERCLOUD), Tekes — the Finnish Funding Agency for Innovation (CloSer project), and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

[6] https://www.arm.com/products/system-ip/coresight-debug-trace

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, pages 4:1–4:40, 2009.
[2] 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 CCS*, 2016.
[3] J. Backer, D. Hely, and R. Karri. On Enhancing the Debug Architecture of a System-on-Chip (SoC) to Detect Software Attacks. In *IEEE DFT*, 2015.
[4] K. A. Bailey and S. W. Smith. Trusted Virtual Containers on Demand. In *ACM-CCS-STC*, 2010.
[5] A. Basak, S. Bhunia, and S. Ray. A Flexible Architecture for Systematic Implementation of SoC Security Policies. In *ACM/IEEE DAC*, 2015.
[6] A. Basak, S. Bhunia, and S. Ray. Exploiting Design-for-Debug for Flexible SoC Security Architecture. In *ACM/IEEE DAC*, 2016.
[7] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *ACM/IEEE DAC*, 2015.
[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security*, 2005.
[9] R. d. Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. d. Bosschere, B. Preneel, B. d. Sutter, and I. Verbauwhede. SOFIA: Software and Control Flow Integrity Architecture. In *ACM/IEEE DATE*, 2016.
[10] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. In *ACM CCS-STC*, 2009.
[11] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *ISOC NDSS*, 2012.
[12] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *ACM CCS*, 2008.
[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 S&P*, 2016.
[14] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Chapter 36 Intel Processor Trace. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2016.
[15] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *IEEE/IFIP DSN*, 2009.
[16] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *USENIX OSDI*, 2014.
[17] L. Le. ARM Exploitation ROPMap. BlackHat USA, 2011.
[18] Pulpino. An Open-Source Microcontroller System based on RISC-V. https://github.com/pulp-platform/pulpino.
[19] RISC-V. The Free and Open RISC Instruction Set Architecture. https://riscv.org/specifications, 2016.
[20] A.-R. Sadeghi and C. Stüble. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *NSPW*, 2004.
[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security*, 2004.
[22] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based Attestation for Embedded Devices. In *IEEE S&P*, 2004.
[23] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM CCS*, 2007.
[24] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting Hardware Advances for Software Testing and Debugging (NIER Track). In *ACM/IEEE ICSE*, 2011.
[25] J. Viega and H. Thompson. The State of Embedded-Device Security (Spoiler Alert: It's Bad). *IEEE S&P*, 10(5):68–70, 2012.
[26] X. Wang, Y. Zheng, A. Basak, and S. Bhunia. IIPS: Infrastructure IP for Secure SoC Design. *IEEE Transactions on Computers*, Aug 2015.