LAHEL: Lightweight Attestation Hardening Embedded Devices using Macrocells

Orlando Arias
University of Florida
Email: orlandoa@ufl.edu

Dean Sullivan
University of Florida
Email: deanms@ufl.edu

Haoqi Shan
University of Florida
Email: haoqi.shan@ufl.edu

Yier Jin
University of Florida
Email: yier.jin@ece.ufl.edu

***Abstract*—In recent years, we have seen an advent in software attestation defenses targeting embedded systems which aim to detect tampering with a device's running program. With a persistent threat of an increasingly powerful attacker with physical access to the device, attestation approaches have become more rooted into the device's hardware with some approaches even changing the underlying microarchitecture. These drastic changes to the hardware make the proposed defenses hard to apply to new systems.**

In this paper, we present and evaluate LAHEL as the means to study the implementation and pitfalls of a hardware-based attestation mechanism. We limit LAHEL to utilize existing technologies without demanding any hardware changes. We implement LAHEL as a hardware IP core which interfaces with the CoreSight Debug Architecture available in modern ARM cores. We show how LAHEL can be integrated to system on chip designs allowing for microcontroller vendors to easily add our defense into their products. We present and test our prototype on a Zynq-7000 SoC, evaluating the security of LAHEL against powerful time-of-check-time-of-use (TOCTOU) attacks, while demonstrating improved performance over existing attestation schemes.

I. Introduction

Embedded systems are pervasive and increasingly interconnected as society adopts an expanding IoT and CPS infrastructure. Many of these devices run bare-metal firmware with strict real-time requirements, which leaves little room for integrating software security solutions hardened against runtime attacks. An equally robust alternative to protecting systems against runtime attacks is to ensure the runtime integrity of the embedded firmware. Devices typically provide two defenses to ensure integrity: 1) lock-down access to code and data memory so that the firmware can not be replaced or reused; and 2) check that the intended firmware is executing using device attestation.

In this work we focus on the latter and introduce LAHEL, a lightweight IP-based runtime attestation defense. LAHEL improves upon the performance, power consumption, design, and robustness compared to several recently proposed solutions [1], [2], [3], [4], [5].

Whereas previous runtime attestation schemes either interrupt execution via a verifier's authentication request, or make calls into dedicated authentication routines upon a control-flow transfer, LAHEL parallelizes execution and authentication. For our prototype, LAHEL uses the real-time tracing features of the Program Trace Macrocell (PTM) available as part of the CoreSight debug architecture combined with a dedicated program code verification unit to separate attestation from normal execution. Parallelization of runtime hash measurements guarantees equivalent performance to the baseline architecture without sacrificing security.

LAHEL avoids impractical modifications to the CPU core by leveraging available CoreSight components and acting as an AMBA AXI IP core master. An IP-based defense is ideal for SoC vendors because they often are prevented from modifying the CPU due to the core's license agreement. Assuming dedicated secure functionality is offered directly on the SoC package, it would suffer inherently from the inability to be patched in the event of failure. Our prototype LAHEL is built upon FPGA fabric to show how it can be integrated into SoC that provide the tracing facilities we utilize. Finally, LAHEL benefits chip manufacturers by avoiding modifications to the CPU and the associated non-trivial changes to subsystems on the critical path such as the front-end, branch-predictor, and cache subsystem.

LAHEL attests not only dynamic control-flow, but also the instructions executed within each basic block to ensure security against physical attackers attempting to change program code as a time-of-check-time-of-use (TOCTOU) attack [4]. In approaches such as SMART [1], which statically checks a region of memory upon an attestation request, or C-FLAT [2] and LO-FAT [3], which check a program's control flow as it occurs, it is possible for a physical attacker to execute malicious code in between either defenses request for an attestation quote. LAHEL eliminates this vulnerability by closing the gap between requesting an attestation quote and its verification.

In short, the contributions of this work are:

1) We design a novel IP-based attestation defense resilient against runtime attacks launched by physical attackers. LAHEL is built as an IP core that can be integrated into existing SoC designs without changing the CPU core.
2) We show how real-world metrics affect our design choices in order to obtain a design which is capable of providing the same security guarantees as existing mechanisms while also improving on their capabilities.
3) We evaluate LAHEL in terms of area, power, and performance cost to software. We show how the real-time tracing facilities of the PTM provide no overhead when active, the effects of reading program code from our

core, as well as the demands of the proposed attestation scheme in terms of hardware and software design.
4) We evaluate the security of LAHEL against strong physical attackers capable of launching TOCTOU attacks and show that it can prevent them.

The remainder of this paper is structured as follows. Section II establishes an introductory background to the problem of device attestation, as well as the components we use in LAHEL. Section III presents the attack model we will follow in the paper. Section IV introduces the architecture and implementation of LAHEL. We evaluate our design in Section V, then discuss some of our observations in Section VI. We describe related works in Section VII before presenting concluding remarks in Section VIII.

II. Background

A. Device Attestation

Device attestation is a method to determine whether a device is operating as intended or not. In an attestation mechanism, a *prover* $\mathcal{P}$ performs a measure of the device. The measure is then checked and validated by a *verifier* $\mathcal{V}$, which compares it to a series of known measures to determine if the device is operating properly.

The challenge posed by an attestation approach is that of the design and functionality of $\mathcal{P}$. Being part of the device under check, $\mathcal{P}$ must be resilient against a wide variety of attacks. An adversary aiming to tamper with the device must be able to either bypass or feign the functionality of $\mathcal{P}$ to remain undetected. As such, $\mathcal{P}$ must be resilient to these avenues of attack. Previous approaches [1], [2], [3] have accomplished this task with variable success and different arrays of compromises [4], [5].

The characteristics exhibited by the implementation of $\mathcal{P}$ define whether an attestation mechanism is *static* or *dynamic*. *Static* attestation mechanisms are those where $\mathcal{P}$ interrupts normal operation of the device to perform a measure. *Dynamic* attestation mechanisms are those where $\mathcal{P}$ is capable of performing measures on the device without interrupting its operation.

We formally define attestation similar to [6]. We let $\mathbb{A}$ be the possible responses of $\mathcal{V}$. That is, $\mathbb{A} = \{0, 1\}$ with 1 representing the device being tested to be functioning properly and 0 an attestation failure. Let $\mathbb{M}$ be the set of measures returned by $\mathcal{P}$. The operation of $\mathcal{V}$ is given by the mapping $V_{k,n} : \mathbb{M} \to \mathbb{A}$. $\mathcal{P}$ generates $\mathcal{M} \in \mathbb{M}$ by performing a computation over the $\mathcal{S}$ of the device, where $\mathcal{S}$ can be any identifiable quality of the device. We alow $\mathbb{S}$ to represent the possible set of states of the device, which may contain illegal ones on compromised units. Then, $\mathcal{P}$ performs the operation $P_{k,n} : \mathbb{S} \to \mathbb{M}$. Then, attestation is defined as follows.

Definition 1: Device Attestation [6]

Let $\mathcal{S}$ be a state of a device. Let $P_{k,n} : \mathbb{S} \to \mathbb{M}$ be the prover operation of mapping a $\mathcal{S}$ to a $\mathcal{M}$. Let $V_{k,n} : \mathbb{M} \to \mathbb{A}$ be the verifier operation of determining whether $\mathcal{M}$ corresponds to a valid state of the device. Then, we define *device attestation* to be the operation of computing $V_{k,n}(\mathcal{M})$.
We further define an *operational device* if and only if

$$\forall \mathcal{S} \in \mathbb{S}, \mathcal{M} = P_{k,n}(\mathcal{S}) \in \mathbb{M}, V_{k,n}(\mathcal{M}) = 1. \quad (1)$$

Conversely, a device is deemed *compromised* or *inoperable* if

$$\exists \mathcal{S} \in \mathbb{S}, \mathcal{M} = P_{k,n}(\mathcal{S}) \in \mathbb{M}, V_{k,n}(\mathcal{M}) = 0. \quad (2)$$

We should note that any $\mathcal{M}$ transmitted by $\mathcal{P}$ must be both unforgeable and non-replayable. If an attacker is capable of forging a $\mathcal{M}$, then $\mathcal{V}$ can be tricked into thinking the device is functioning properly. Moreover, if $\mathcal{V}$ accepts two equal $\mathcal{M}$, then an attacker can just present the same valid $\mathcal{M}$ multiple times for a successful attestation request. That is to say, given a particular $\mathcal{S}_i, \mathcal{S}_j, \mathcal{S}_l ... \in \mathbb{S}$ collected at different times, with $\mathcal{S}_i = \mathcal{S}_j = \mathcal{S}_l = ...$, $\mathcal{P}$ must return $\mathcal{M}_i, \mathcal{M}_j, \mathcal{M}_l, ... \in \mathbb{M}$ so that $\mathcal{M}_i \neq \mathcal{M}_j \neq \mathcal{M}_l \neq ....$ For these purposes, we introduce a pre-shared key k and a nonce n in the formalization above.

B. ARM CoreSight Components

The ARM CoreSight is a series of IP cores that allow for different mechanisms of debug and trace to be performed in ARM-based System on Chips (SoCs). SoC vendors are able to license separate subsystems from the CoreSight components to provide an infrastructure capable of debugging, monitoring, and optimizing the performance of their product [7]. CoreSight allows for execution and data traces to be collected by the CPU and to be transmitted outside CoreSight implementations are divided into three components: *sources*, a *link*, and *sinks*.

CoreSight sources are components which capture execution traces from the CPU or other hardware-related events. The *Program Trace Macrocells* (PTM), which captures control-flow information by tracing waypoint instructions; the *System Trace Macrocell*; and the *Embedded Trace Macrocells* (ETM), which can collect both control-flow and data-flow information from a CPU, are examples of sources in the CoreSight architecture. A system may implement multiple sources, and multiple sources can be active at the same time.

Traces collected from the sources must be grouped into a single stream before being processed. This is where the *link* comes into play. The link takes traces from different sources and combines them into a single stream. Trace sources can be given priority when collected by the link interface, as well as given a source ID for later identification. The *CoreSight Trace Funnel*, which serves as the primary collector for trace information; and the *replicator*, which performs a "live copy" of the capture traces, are examples of link interfaces.

The combined trace streams are then sent to *sink*s. Sinks collect information from the funnel and allow it to be routed out of the SoC through a high speed bus, or store it in a special memory within the SoC. Examples of sinks include the *Embedded Trace Buffer*, an embedded memory that is accessed through a set of memory mapped registers; the *Embedded Trace Router*, which redirects trace output onto the AMBA AXI bus making it available to any AXI master or through the ETR itself as if it was an ETB; and *Trace Port Interface Unit*, which serves as an off-chip drain for traces.

C. The Xilinx Zynq-7000 SoC

The Zynq-7000 SoC is an FPGA designed and manufactured by Xilinx Corporation which contains a dual core ARM Cortex-A9 processor surrounded by a few hard peripherals, such as a DDR controller, and a QSPI controller, as well as reconfigurable fabric. Xilinx terms the ARM cores as the *processing system* (PS) and the FPGA side the *programmable logic* (PL). The PL can be reconfigured directly from the PS, by loading it from an externally connected storage device, or a standard JTAG probe. The PS and PL can interface with each other via AMBA/AXI controllers which can be used as either master or slaves in the bus topology.

The Xilinx Zynq-7000 SoC offers a CoreSight implementation featuring two Program Trace Macrocells (PTM0, PTM1), an Instrumentation Trace Macrocell (ITM), an Embedded Cross Trigger (ECT), a Xilinx Fabric Trace Monitor (FTM), a CoreSight Trace Funnel (ETF), a CoreSight Replicator, an Embedded Trace Buffer (ETB), and a Trace Port Interface Unit (TPIU) [8].

III. Attack Model

Before describing the general overview of our system, we first discuss our adversarial model. Our aim is to defend an embedded IoT device against a malicious attacker. We assume that the device provides the means of establishing a root of trust, and that this root of trust is safe from vulnerabilities. This assumption in line with previous work [1], [2], [3].

The attacker's goal is to change the functionality of the device by altering the functionality of its software. An attacker may accomplish this by maliciously altering control-flow of the device's software, or attempting to change the underlying code. The attacker may attempt to subvert the attestation mechanism by exploiting its operation. We now discuss ways this can be done.

A. Code-Reuse Attacks

Code-reuse attacks (CRAs) are a powerful exploitation mechanism where benign code in a device is used for malicious purposes. CRAs are preformed by combining short instruction sequences, called *gadgets*, from various points in code to generate a new malicious program. These sequences usually end in a control-flow instruction which is used to branch into the next gadget in the series. The usable set of gadgets in a program is called a *gadget catalog*.

Before launching a CRA, an attacker must examine the binary for potentially useful gadgets, building the gadget catalog. Then, an attacker uses a memory vulnerability to corrupt code pointers in order to change the application's intended control-flow. Common targets for corruption are return addresses stored in the stack, as used in *return-oriented programming* (ROP) attacks, or vtable pointers, as used in *jump-oriented programming* (JOP) attacks.

Of importance in a CRA is the ability of the attacker to build a usable gadget catalog. CRAs can be mitigated by reducing an attacker's ability to find potential gadgets. Control-Flow Integrity (CFI) defenses aim towards this goal, by enforcing an application's control-flow graph. However, traditional CFI defenses operate under the assumption that application code is immutable, meaning that they can not be readily utilized under our attacker model.

B. Time-of-Check-Time-of-Use Attacks

A Time-of-check-time-of-use (TOCTOU) attack is a type of attack which is possible on defenses where distinct time periods are used for checks and usage of a particular entity, such as code. An attacker can use the delay between checks and usage to deploy a payload. These mechanisms have been used before to break into devices such as cellular phones and gaming consoles [9].

Under a TOCTOU attack, an attacker waits for the checking mechanism to finish authenticating software. Then, after the check finishes, the attacker switches the software's code or data altering the device's functionality. This type of attack is commonly performed by using so-called *modchips* on devices. Zeitouni et al developed attacks against static attestation mechanisms in [4], proposing ATRIUM as a solution. In this work, we revisit ATRIUM and other dynamic attestation mechanisms and analyze its scalability issues. Our aim with LAHEL is to eliminate these scalability problems while still achieving equal levels of security.

IV. LAHEL Architecture and Implementation

To fulfill the task of dynamic attestation, the proposed LAHEL will attest code dynamically as it is executed by the CPU in order to prevent TOCTOU attacks. A further goal for LAHEL is that it must be efficient both in terms of power consumption and performance overhead. We also avoid modifying commercially available CPU IPs as to provide a solution that works with off-the-shelf components.

For this purpose, LAHEL leverages the Program Trace Macrocell (PTM) IP available in commercial ARM processors. As previously discussed, the PTM allows for high-speed, low-power debugging of instruction and data traces. We utilize the PTM in order to obtain an instruction trace as the CPU executes code. The instruction trace is forwarded to the TPIU, allowing us to read and decode it from the programmable logic using the EMIO interface. The attestation core utilizes the decoded instruction trace to check program code as well as control flow.

We aim for the attestation mechanism to be local. That is, the resulting prototype can attest the device it runs on and notify a remote party of any detected failures. This allows us to reduce storage demands on the device, as well as simplifying the protocol of communication between the device and a remote verifier to only include a periodic heartbeat, alongside reports of device status. The heartbeat protocol in our implementation is based in the one presented in [10].

A. Architecture Overview

We present a high level overview of LAHEL in Figure 1. LAHEL makes use of the TrustZone extension in the ARM architecture to provide a root of trust to perform local attestation using a hardware module. The root of trust provides system bootstrapping and low-level handling of the hardware. As the CPU executes code, our IP core collects instructions in conjunction with the execution trace. When enough instructions are collected LAHEL prepares them to be verified. Because the verification operation and instruction fetching are disjoint operations, the two can be done in parallel. This parallelism allows for the CPU to keep executing instructions with LAHEL's core receiving them alongside new program traces.

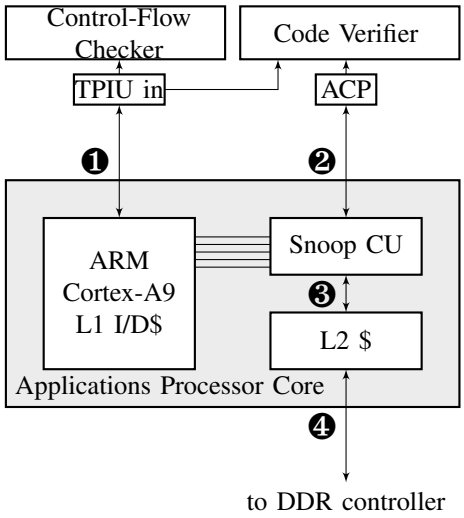

Fig. 1. High level overview of LAHEL. Our IP core interfaces with the TPIU, ❶, to extract control-flow information. TPIU traces are decoded and forwarded to the control-flow checker and code verifier. Program code is read through an AMBA AXI master connected to the Snoop Control Unit, ❷, which reads code from the L2 cache ❸, forwarding any misses to the DDR controller, ❹.

LAHEL utilizes the obtained control-flow information and program code to securely check whether the application is behaving normally. A mismatch to the expected behavior raises an interrupt. The root of trust determines the cause of failure, creates a failure report and sends it to a trusted party, and finally attempts to restore the device to known working behavior.

Once integrated, LAHEL is deployed in a two step process. First, an offline phase is performed in order to collect basic block and control flow information. Basic block information is obtained and securely stored in the device. The encoded control-flow metadata is stored within the unit itself. At this point, the device is ready for the second phase: on-field deployment. The device is connected to a network and the initial handshake with the remote verifier occurs. As the device operates, it self attests and responds to heartbeats from the remote verifier. Using the on-board metadata, if any faults are detected the device notifies the remote verifier with the fault condition so that proper action can be taken.

B. Bootstrapping System

The Cortex-A9 cores in the Zynq-7000 SoC boot in secure mode by default. We utilize this behavior to set up the PTM, ETF, TPIU, and our IP core and isolate them from the rest of the system before dropping execution privileges to the normal world. This hardware-enforced separation prevents normal world software from reconfiguring the components necessary to achieve our security requirements.

We restrict program tracing to the areas covered by the normal world. For this purpose, we modified the linker script to declare a section of memory to store the secure world code, and a section of memory to store the normal world code. The Zynq-7000 memory controller allows configurations of TrustZone regions on $64\,\mathrm{MiB}$ regions. A region this size is sufficient to hold both code and data for the secure bootstrapping process, while the other regions are used for regular code and data

CoreSight components will only send data whenever a full packet is filled. Since we wish to obtain trace data as fast as possible, we configure the TPIU interface to be $8\,\mathrm{bit}$ wide. This allows the smallest packet, an atom, to be sent when filled. Wider packets, such as `i-sync` packets, are sent one byte at a time.

C. Capturing and Decoding Instruction Traces

As previously mentioned, we configured the TPIU to interface to be $8\,\mathrm{bit}$ wide. We expose the TPIU to the programmable logic using the EMIO interface. We only utilize three of the four TPIU signals in our design. This is because the TPIU clock output is a double data rate clock running at half the frequency of the input clock. ARM stating the TPIU data output to be sampled at both the rising and falling edges. However, both the TPIU data and TPIU input clock are synchronized. As such, we are able to drive our data capture with the input clock, simplifying our design. We also enable branch broadcasting in the PTM. This has the effect of generating a branch packet whenever a waypoint instruction generates a control-flow event, whenever a branch instruction is taken. If a branch instruction is not taken, the behavior is recorded within an atom packet. The PTM will not send the atom packet until it is full, or until a different event triggers the system.

Since we target single-threaded systems, we disable formatting in the TPIU output[1]. TPIU data is only valid whenever the `tpiu_ctl` is low. Furthermore, the TPIU may send multibyte packets which need to be reassembled. For this purpose, we utilize a state machine that captures TPIU data and decodes resulting PTM stream.

The state machine waits for the initial `i-sync` packet indicating the start of a trace. Embedded in the `i-sync` packet is the start address of the trace which we store in a ring buffer for processing by the program code verification subsystems. We also store this address in the state machine. After the `i-sync` packet, the TPIU will send either atoms or branch packets depending on the control-flow transitions of the software. When receiving a branch packet, we utilize the stored address to decompress it. We store the newly decoded address as well as send it to a ring buffer for storage and processing. Because branch broadcasting is enabled, atom packets only contain information about branch instructions that were not taken. We decapsulate the atom packet and obtain a count of the number branch instructions that are not taken and store this total in the ring buffer. Since addresses in the ARM architecture are always even value, we differentiate the branch not taken counts from addresses by storing $n \times 2 + 1$ in the ring buffer, where n is the number of control-flow instructions that were not taken.

Due to the amount of data going into the ring buffer, we do not store repeated atom packets. We store repeated branch addresses since these may be due to indirect control-flow. We are unable to identify whether the branch was caused by an indirect control-flow instruction or not when branch broadcasting is enabled. When branch broadcasting is disabled branch packets are only generated whenever an indirect control-flow transfer occurs. Control-flow transfers due to direct branches are encoded within atom packets. However, the PTM will not send an incomplete packet, meaning that code verification will be delayed until a few basic blocks are executed.

1) Example Trace Decoding: We utilize the micronenchmark in Listing 1 as a way to test our trace decoding facilities. The microbenchmark presents a tight loop which iterates 2048 times. The microbenchmark continuously subtracts the value 1 from a register, first initialized to 2048, while setting the flags register. We repeat this process until the register reaches the value of 0. This results in the loop terminating and all subsequent branch instructions not being taken. Consequently, we expect our trace to be mainly composed of 2048 branch packets, and one to three atom packets signaling a branch not taken. The captured trace is shown in Figure 2.

```
uint32_t i = 2048;
enable_ptm();
__asm__ volatile(
    "1: subs %[input], %[input], #1"  "\n"
        /* decrement register */
    " bne 1b"                         "\n"
        /* if not zero, loop */
    " bne 1b"                         "\n"
        /* branch never taken */
    " bne 1b"
        /* branch never taken */
  :
  : [input] "r"(i)
  : );
```

Listing 1. Microbenchmark used to obtain a sample trace.

```
00 00 00 00   00 80 08 f0   0e 10 00 21   86 87 12 09
09 09 09 09   09 09 09 09            ...
09 09 09 09   09 09 09 9e   e7 07 01 00
```

Fig. 2. Trace result for microbenchmark. We see an `a-sync` packet followed by a `i-sync` packet indicating the start of the trace. An `atom` packet proceeds this, followed by a series of `branch` packets, an `atom` packet, a `branch` packet, and the end of trace.

We start with an `a-sync` packet. This packet is constituted by at least five `00` bytes followed by one `80` byte. We then encounter an `i-sync` packet (`08 f0 0e 10 00 21`) detailing the address at which the trace started. This is followed by an atom packet, `86`, and a series of branch packets `87 12`. These packets come from the end of the `enable_ptm()` function. The ensuing branch packets are generated by our loop.

D. Checking Program Code

We connect an AMBA AXI master to the Accelerator Coherency Port in the Zynq-7000 SoC. This allows us to directly access the Snoop Control Unit (SCU). Accesses initiated by the AXI master are serviced by the SCU. The SCU can read directly from the L2 cache within the processor cores. If the requested instructions are not in the cache, the SCU forwards requests over the DDR controller, resulting in the instructions being fetched from memory.

We designed our AMBA master to read in bursts of 16 B, as this corresponds to the size of our code checking algorithm. Once the read completes, we determine if any control-flow instruction is found within it. If the block contains one, any instruction after the basic block is replaced with padding. We send the padded basic block to the hash engine for compression. If the block contains no control-flow instruction we add no padding and send a signal to the verification engine to chain the next basic block.

We also report whether a control-flow instruction is indirect, in which case we compute its address and forward it to the control-flow checking submodule. This is used as the source address when verifying control flow. Section IV-E describes why we only track indirect branches in control-flow.

E. Checking Control-Flow

As part of our attestation mechanism, we also verify the program's control-flow. Most of the previous hardware-based CFI approaches use some form of code instrumentation to ensure the application is executing correctly [11], [12], [13].

[1]With formatting enabled we need to wait for 16 B to be received from the TPIU interface. The last byte contains the least significant bit of all odd bits in the received stream. Not only would our hardware need to wait for the entire stream to be received, it would also need to apply an extra decoding step causing further delays on verifying control flow and obtaining execution addresses.

Since we are unable to add new instructions having no access to the CPU core we employ data returned by the TPIU decoder to monitor control-flow.

The TPIU decoder provides us with the target address of a control-flow instruction if control-flow was altered, or a series of atoms in the case it was not. We use a mechanism similar to the one proposed in [14] with a few crucial differences. The work in [14] does not postulate how control-flow metadata is stored. We now examine a few possibilities.

We utilize the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [15] as a dataset to examine control-flow behavior. The BEEBS suite are designed to test the performance of deeply embedded systems targeting energy consumption. Programs in the suite include image encoders, implementation of compression and cryptographic algorithms, and graph navigation routines, among others. Because of its wide coverage, we believe the BEEBS suite is representative of code running on embedded devices.

We first note that direct calls have the offset to the target address encoded in them. Attackers wishing to redirect control-flow using these instructions as targets must overwrite the instruction stream. Since we handle modifications to program code as a separate mechanism (Section IV-D), we do not need to track these instances.

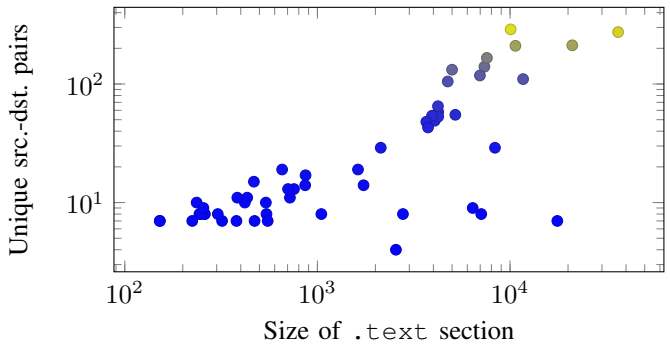

Fig. 3. Distribution of unique indirect control-flow transfer address pairs in the BEEBS benchmark suite compiled for the ARMv7-A architecture. We only include the integer subset of the benchmarks in our analysis. Root of trust routines are not included in this plot.

We use `cflow` [16] to find an estimate of the control-flow graph of the code. We then use the compiled code to find the number of return sites on every function. With this information, we find the number of unique {source,destination} pairs. We also look for other indirect control-flow transfers, such as those generated by indirect calls to functions. Our results are shown in Figure 3. Here, we plot the number of unique indirect control-flow transfers versus the code size of the binary. We do not count any bootstrapping code, as this belongs in the root of trust, which is assumed to be secure. Our largest benchmark is measured at 36 376 B for `miniz`, while the most indirect control-flow transfers is measured at 289 on `pico_jpeg`. For the following analysis we will round up these numbers to the next power of two ($36376 \rightarrow 65536$, $289 \rightarrow 512$).

Using these figures, we can generalize that a $16\,\mathrm{bit}$ source address can target any given $16\,\mathrm{bit}$ target address. We can reduce this number in the ARM architecture under the Aarch32 model, since instructions must be $32\,\mathrm{bit}$ aligned. As such, we effectively reduce control-flow transfers from a $14\,\mathrm{bit}$ source address to a $14\,\mathrm{bit}$ target address. We can then create a $2^{28} \times 1$ memory to hold control-flow transfers. The memory is addressable using the concatenation of the source and target addresses in the control-flow transfer, and preconfigured with values indicating valid control-flow transitions. Whenever an indirect control-flow transfer occurs the system checks the contents of the memory. If the address contains a 1, the transfer is valid and the check succeeds, otherwise the check fails and the control-flow subsystem triggers an error. However, this approach is largely wasteful. Of the $256\,\mathrm{Mbit}$ memory we created we only utilize $1.19 \times 10^{-5}\%$ of its capacity to store 512 control-flow transfers.

We can attempt to improve this model by using a different hardware construct. We observe that for our benchmark suite, no more than 512 control-flow transfers need to be stored. We can then employ a content-addressable memory (CAM) capable of holding 512 different entries. Lookups to this CAM are done using the source-target address pairs. The CAM itself does not need to store any information regarding control-flow data. Returning whether the control-flow transfer is stored is sufficient. This reduces the memory requirements to $28 \times 512 = 14\,336\,\mathrm{bit}$. However, it introduces the overhead of a CAM. We implemented this CAM to test for overhead on our FPGA obtaining $\approx 13.47\%$ and $\approx 16.37\%$ flip-flop and LUT utilization, respectively, with an estimated power consumption of $70\,\mathrm{mW}$.

Instead of these approaches, our design reduces complexity and space by utilizing a Bloom filter [17], [18]. A Bloom filter is a probabilistic data structure which has the property of being space efficient while allowing to test whether an element is a member of a set. For a Bloom filter with m bits, n stored elements, and k different hash functions with no significant correlation with each other, the optimal number of bits per element for a false positive probability p is given by

$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2}, \tag{3}$$

and the corresponding number of hash functions is approximately

$$k = -\log_2 p. \tag{4}$$

If we size our Bloom filter to contain the same number of memory elements as the previously described CAM, we obtain a false positive rate of $1.437 \times 10^{-4}\%$ with 20 independent hash functions. If we size our memory to the next available power of 2, we decrease the false positive rate by over one order of magnitude while increasing our hash function count to 22. In this event, each hash function needs to output a $14\,\mathrm{bit}$ number. Using the hashing algorithm described in [19], our Bloom filter utilizes a total of $\approx 3.94\%$ of the LUTs in our

FPGA, with $\approx 6.63\%$ of the available DSP slices. It exhibits a power consumption of $28\,\mathrm{mW}$.

The issue with this approach is that our storage element is about 25% of the size required to store the largest of our benchmarks. To solve this particular predicatment, compression algorithms such as Golomb-Rice coding [20], [21] can be employed to reduce the amount of memory consumed. We forgo compression on the Bloom filter as compression assumes a sparsely populated memory (which holds during our testing but may not be true for some other cases), and in order to simplify the underlying logic. Using Equation (3) we see that we can generate an $8\,\mathrm{KiB}$ storage element for the bloom filter and still achieve a 0.459% false positive rate. We discuss the implications of this number in Section V.

F. Asynchronous Events

Interrupt handling results in an asynchronous control-flow event, where the currently executing instruction finishes then control-flow is redirected to an interrupt handler. In the ARM architecture interrupts can be nested. That is, an interrupt with higher priority will preempt the execution of a handler for an interrupt with lower priority. Fortunately for us, the PTM generates an `i-sync` packet whenever control-flow changes due to an interrupt. We process this `i-sync` packet to extract the address of the interrupt handler. The PTM does not give us the address of the next instruction that would have been executed. However, the AXI master performing code attestation reads from the ring buffer atom packets and branch packets, deducing which basic blocks are being executed. Upon detection of the `i-sync` packet generated by the PTM, we are able to determine the basic block which was executing once the interrupt triggered. We utilize this behavior to estimate possible return sites for the interrupt handler. We should note that interrupt service routines are allowed to change the return address. Notable cases are as results of scheduling or thread creation. These events are not applicable to our platform given that we target single threaded embedded devices.

V. Evaluation

A. Control-Flow Evaluation

Because our control-flow checks are based on a probabilistic data structure, there is a real possibility that control-flow is violated and the LAHEL core does not detect it. With a false positive rate of 0.459% and a total of 512 indirect branches stored in the Bloom filter, we see that each indirect branch has ≈ 2.35 return sites that will trigger a false positive. These invalid return addresses depend on the hash functions used in the Bloom filter, as well as the control-flow behavior of the program, as this determines the actual contents of the Bloom filter.

An attacker may attempt to find locations where these false positives are triggered. If using a cryptographically secure hash function to index the bloom filter the process becomes unfeasible. However, the hash function we utilized does not exhibit this property and we are left with an attacker who is able to find collisions with some effort. We argue that in the event of a collision, the number of viable targets make a code-reuse attack very limited in scope. Code-reuse attacks work utilizing a gadget catalog, with a gadget being a small unit of code that an attacker can use for malicious purposes (e.g. setting a value in a register before jumping to a location which uses that value for nefarious purposes).

Our control-flow policy has the limitation that it does not keep state. That is, multiple *valid* return sites are allowed for function returns. An attacker with the ability to corrupt the return address of a function can redirect control to a different place other than the current caller. This is called a control-flow bending attack [22]. During our testing we were unable to identify a source of collisions that would result in a "good" gadget catalog to perform a code reuse attack. The available gadgets were either too disjointed or too few to significantly alter the behavior of the program in a way that did not result in a hard crash.

B. Program Verification Evalualion and TOCTOU Resiliency

Previous attestation mechanisms utilize a hash function for the purpose of verifying the code being executed. In our scenario, the code being verified is organized in basic blocks and is fetched through the acceleration coherency port. ATRIUM [4] stores the computed hashes locally before sending them to a verfier on an attestation request. The specifics of the storage capabilities, the frequency at which the hashes are generated, and the frequency of the incoming attestation requests are not specified. The higher the hash generation frequency, and a lower frequency of attestation requests yield for a larger storage requirement. LiteHax [5] aims to subvert the storage requirement problems by streaming the control-flow and data-flow hashes to the verifier as they are generated. This puts greater pressure on the network, as it must have the necessary throughput and reliability to transmit the hash information.

We now consider a hash function h as a collision-resistant one way compression function. The former characteristic means that for two inputs it is infeasible to generate $h(x) = h(y)$ for $x \neq y$. The latter specifies that given a digest d, it is infeasible to find x such that $h(x) = d$. Hash functions used in attestation serve one purpose: to securely generate a measure of the device without the need to transfer large amounts of data. For example, in SMART [1] the prover securely generates a digest of a range of program code and sends it to a remote verifier as a response to an attestation request. In doing so, the amount of information transferred over the network is reduced: instead of transmitting the code being verified, a representation of that code is sent instead in the form of the computed hash. In the case of ATRIUM, basic blocks are hashed and the resulting digest is stored in a secure memory.

We examine the compiled BEEBS binaries and extract basic block information. We count the number of instructions per basic block and determine averages showing our results in Figure 4. We note that basic blocks contain three instructions as an average. This gives us insight into the compression

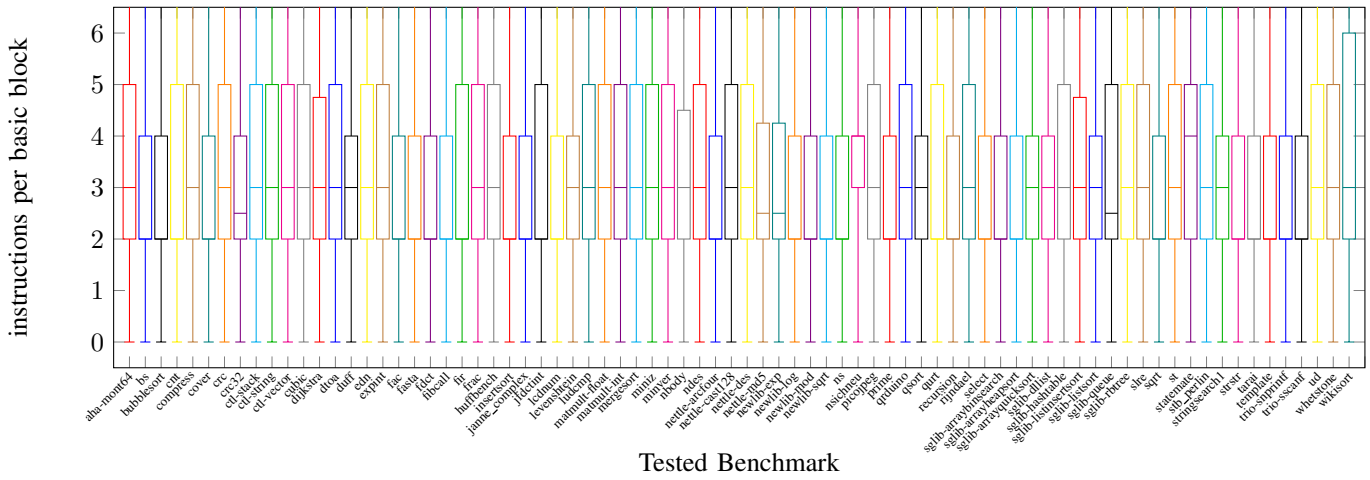

Fig. 4. Average instruction count per basic block in BEEBS. We exclude outliers in our plots, which we define to be 2.5 standard deviations away from the mean value of instructions per basic block for a particular benchmark. Benchmarks exhibit an average of three instructions per basic block. This implies that under the ARM ISA a basic block is 12 B (96 bit) in size.

characteristics we desire of our hash function. For a remote attestation mechanism with the characteristics of ATRIUM, hashes are collected with the objective of sending them at a later time to a remote verifier. An attacker is prevented from tampering with the hashes by saving them in a secure storage. However, the hash functions which we consider to be cryptographically secure today produce a digest of at least 128 bit. Specifically, SHA-3, used in LiteHax, generates a digest of 224 bit, and BLAKE2's digest in ATRIUM is of 512 bit. This is considerably larger than the measured average block input of 96 bit. Considering that we must store these hashes in a secure place for either ATRIUM or LAHEL, and that the storage demands for this storage is higher than that of program code itself we must question whether it is desirable for the approach to actually hash basic blocks.

For our purposes, we store a shadow copy of the valid program code instead of its internal hashes and compare against it during execution. This has two advantages: our storage requirements are smaller than the one requiring a hash engine, and our root of trust is able to restore the firmware from a known working copy in case the LAHEL detects the firmware has been modified. As such, in the event of detecting a change in program code the root of trust will generate and send a report a remote trusted party then proceed to reprogram the device to a known valid base state.

As described in [4], the execution model of SMART yields a timing signature which reveals when the CPU is accessing program code for an attestation request. LAHEL eliminates this side channel by obtaining the instruction stream from the SCU which reads data directly from the L2 cache and forwards any misses to DRAM. During our testing we traced the signals from the AXI master and measured response latency. We noted a delay corresponding to the access time in L2. This is due to the instruction stream executed by the CPU was moved into cache when it performed instruction fetches. An attacker wishing to execute malicious code using the attack presented in [4] would need to force L2 misses when LAHEL attempts to access the cached instruction stream.

VI. Discussion

Our sample implementation revolves around the Xilinx ZYNQ 7000 SoC which contains a Cortex-A9 processor in a dual-core configuration. The Cortex-A9 implements the ARMv7-A architecture with a superscalar, variable length, out-of-order pipeline and dynamic branch prediction [23]. This type of core may not be applicable for some embedded applications we target with our defense. Our implementation both suffers and benefits from having an applications class processor. An applications class processor enables us access to a fully featured Memory Management Unit. As such, we are able to separate the CoreSight components from being accessed by the running software preventing it from disabling the trace functionality.

Note that the superscalar out-of-order pipeline affect the way the CoreSight components behave. This results in an influx of data from the TPIU into our subsystem at unexpected times. We used the microbenchmark in Listing 2 during testing of the TPIU interface. We captured the traces shown in Figure 5 using an instance of the Integrated Logic Analyzer [24]. We observe that the TPIU reports four consecutive branch packets, followed by two cycles where the TPIU reports no data. On a simple pipelined processor we expect to receive the branch packet every few cycles, as the subtract operation and the nop instructions make their way through the pipeline. Instead, due to the out-of-order and superscalar nature of the Cortex-A9 we see the reflected behavior.

```
1: subs r0, r0, #1
            ;; r0 - 1 --> r0, set flags
   nop; nop; nop; nop
   nop; nop; nop; nop
   nop
```

```
6       bne 1b  ;; branch if non-zero result
```

Listing 2. Microbenchmark used to test bandwidth on the TPIU interface.

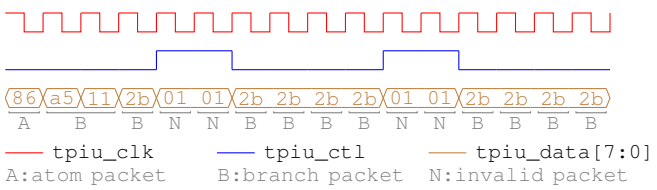

Fig. 5. ILA capture during the execution of the microbenchmark in Listing 2. We observe bursts of consecutive branch packets during execution with a two cycle gap in between due to the out of order nature of the test CPU.

We should note that the PTM never generates branch packets or atom packets due to branch prediction. Only after branch instructions are committed does the PTM generate the necessary packets. As such, we never see the effects of the branch predictor in the TPIU.

A. Migrating to the Cortex-M

The Cortex-M line of microcontrollers are more suited to embedded devices than the application-class processors found in the Cortex-A line. Unfortunately, as far as we are aware there is no publicly available platform with a Cortex-M which we are able to utilize to implement our prototype. The closest we are able to find are the ARM DesignStart FPGA platforms, but these consist of encrypted unmodifiable bitstreams that provide a Cortex-M core and peripherals [25].

The Cortex-M line, however, still provides the CoreSight components as an optional part of the implementation. During our research we found that microcontroller vendors provide the Embedded Trace Macrocell (ETM) as part of their offering, and about half of the ones we encountered provide a TPIU interface. The ETM offers the same functionality as the PTM with the additional benefit that it can also perform data traces. As such, a LAHEL implementation targeting these platforms can potentially be extended to perform data tracing. Furthermore, with the advent of the ARMv8-M architecture we are given an embedded version of the security extensions (TrustZone-M) [26]. The root of trust in LAHEL can be bootstrapped from a TrustZone-M environment, while also locking away the debug peripherals from being accessed by the program.

B. Scaling to Real-Time and Multitasking Operating Systems

Our current implementation of LAHEL targets embedded devices running single threaded applications. However, we believe our system can be extended to include support for real-time and multitasking systems. The TPIU can be configured to issue formatted packets. This allows us to use multiple trace sources concurrently. By enabling the formatter we can include trace IDs in the PTM, and switch these trace IDs dynamically. Then, by modifying the scheduler in a real-time or multitasking operating system, we can set new trace IDs in the PTM and thus we are able to differentiate control-flow events in different programs.

However, when enabling the formatter, the trace must be first decapsulated by waiting for a 16 B TPIU frame to arrive at our decoder. Once decapsulated, we can then decode the trace and check control-flow and program code. For operating systems that use virtual to physical address translation in programs, our core must be further aware of which physical frames are occupied by a running application. Hence, mechanism to perform translation from virtual to physical addresses for the core must also be provided, potentially introducing further delays in our system.

VII. Related Works

In this section we provide a brief description of proposed attestation mechanisms with emphasis on the changes made to the hardware and software stack.

Eldefrawy et al. propose SMART [1] as means of providing software attestation in embedded devices. SMART is a static attestation mechanism that implements an on-chip prover which responds to requests from a remote verifier. The prover is implemented in software and embedded inside a ROM in the microcontroller. The software uses an on-board shared key to hash a verifier-specified portion of software running from an external memory. Although the shared key is memory mapped, SMART adds hardware to ensure it is only accessible whenever the prover ROM code is executing. Furthermore, newly introduced hardware also ensures that the ROM code executes from its proper entry point. This is done to prevent leakage of the key through a code-reuse attack. Although these precautions are taken, a device protected by SMART can still be subject to exploitation through a code-reuse attack outside the attestation ROM as this code is not protected by the ROM mechanism, and control flow is not checked [2]. Furthermore, as demonstrated in [4], an attacker with physical access to the device can multiplex the data bus in order to execute malicious code while the attestation ROM only sees the intended code being executed.

C-FLAT [2] and LO-FAT [3] as means to counter the control-flow vulnerabilities found in SMART. The prover in C-FLAT and LO-FAT perform static attestation by aggregating the execution path of a program, including branches and function returns. These are built into a stream of hashes indicating how the program executed. To service attestation requests from a remote verifier, the prover sends the collected hashes which the verifier uses to ensure the application has executed correctly. The prover relies on a TrustZone environment to protect its code and collected data. Branch instructions in the normal world are replaced with trampolines to the secure world which allows the prover to run. As these approaches check control-flow only, an attacker with access to the device is capable of changing code within basic blocks as long trampoline targets are not changed [4]. Furthermore, because the prover is executed at the behest of the software being attested, an attacker can ultimately leave the prover to collect no control-flow information of compromised code.

Zeitouni et al. propose ATRIUM as means to counter the weaknesses in SMART and C-FLAT [4]. ATRIUM borrows

ideas from C-FLAT and SMART in that both control-flow and program code are checked. However, unlike previous approaches, ATRIUM directly taps into the CPU pipeline to extract instructions as they are executed by the CPU storing them in a buffer. This allows ATRIUM to perform dynamic attestation on the device. The ATRIUM subsystem decodes these instructions to check for control-flow, sending a signal to a loop encoder. Control-flow information is kept by the ATRIUM core. Furthermore, whenever a full basic block is collected, instructions within it are hashed with a BLAKE2b engine. If the instruction buffer is filled while hashing is taking place, the core halts execution in the CPU until the attestation operations are finished. Vendors wishing to implement ATRIUM require direct access to the CPU core to add the necessary components. Under the current licensing model, this is unlikely to happen without the aid of the IP vendors. Furthermore, ATRIUM does not specify the necessary requirements for storage of hashes and execution traces. Lastly, timing critical functions are affected by halting CPU execution resulting in missed events, or delayed response.

LiteHax [5] extends the ATRIUM model by also adding data-flow tracking. Much like ATRIUM, LiteHax requires access to the CPU's pipeline, not only to track execution but also to track data loads and stores. Control-flow, and the results from loads and stores are independently buffered and hashed. Much like ATRIUM, LiteHax performs dynamic attestation on the device as it is capable of gathering the necessary measures for attestation without interrupting the device's functionality. During testing, authors tuned the storage requirements of the buffers and to meet those of OpenSyringePump [27]. This allows for not having to halt execution during the attestation process. Partial attestation reports are streamed to a remote verifier for further analysis and verification. Unfortunately, this requires the hardware to be tuned for the specific application. This is readily doable in soft IPs and made-for-application microcontrollers, but not approachable for general-purpose mass-market devices. Furthermore, the work does not provide analysis on the frequency on which the verifier receives the partial attestation reports, or the necessary network conditions for the verifier to receive a reliable report.

Wahab et al propose an instrumentation mechanism to perform data flow analysis in a program using the CoreSight infrastructure in [28]. The authors instrument a program with a newly created system call to request tracing of an application. Similar to our approach, the proposed mechanism uses the reconfigurable logic in the Zynq-7000 SoC to trace the running application. The authors employ this mechanism to detect a double-free vulnerability in a test application, as well as the reconstruction of an application's control flow graph. Unlike our approach, however, the software must be instrumented with the system calls to start and stop tracing over function calls, which requires binary rewriting if source code is not available.

HardBlare [29] uses the CoreSight system present in ARM cores for information tracking. The authors developed a static analysis pass in LLVM which propagates dependencies to the ARM back-end of the compiler which are stored in the final executable. During execution, an IP core processes trace information generated by the PTM, which propagates the tag information generated during the instrumentation pass. Authors use this mechanism as a way of performing Dynamic Information Flow Tracking (DIFT). Similar to LAHEL, HardBlare does not require modifications to the processor core, only employing new peripherals in the SoC to perform information tracking.

VIII. Conclusion

In this paper we presented LAHEL as a way to demonstrate that dynamic attestation is possible in current CPUs without the need to change them. We demonstrated that a dynamic attestation mechanism can be created using the trace structures of the CoreSight Debug Architecture. We also discussed the implications of using cryptographically secure hash functions as means of data compression when dealing with program code. As future work, we plan on devising means to reduce the necessary TPIU traffic while still being able to maintain trace consistency during the attestation process.

Acknowledgements

We would like to thank the reviewers for their thoughtful mentions on how to improve the paper. The work is partially supported by the Department of Energy through the Early Career Award, the Semiconductor Research Corporation (2018-TS-2860), and Cisco. Mr. Orlando Arias is supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1144246 and DGE-1842473. The views and opinions stated in this work do not necessarily reflect those of the aforementioned funding agencies.

References

[1] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: Secure and minimal architecture for (establishing dynamic) root of trust." in *NDSS*, vol. 12, 2012, pp. 1–15.

[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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 743–754.

[3] 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," *arXiv preprint arXiv:1706.03754*, 2017.

[4] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 384–391.

[5] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[6] O. Arias, F. Rahman, M. Tehranipoor, and Y. Jin, "Device attestation: Past, present, and future," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 473–478.

[7] *CoreSight Technology System Design Guide*, ARM Limited, June 2010, issue D.

[8] *Zynq-7000 All Programmable SoC Technical Reference Manual*, XILINX Corporation, December 2017, v1.12.1.

[9] E. DeBusschere and M. McCambridge, "Modern game console exploitation," *Technical Report, Department of Computer Science, University of Arizona*, 2012.

[10] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "Darpa: Device attestation resilient to physical attacks," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2016, pp. 171–182.

[11] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 74.

[12] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 38–49.

[13] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[14] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of rop/jop monitoring ips in an arm-based soc," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 331–336.

[15] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[16] GNU Project, "GNU cflow."

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[18] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.

[19] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A reliable randomized algorithm for the closest-pair problem," *Journal of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997.

[20] S. Golomb, "Run-length encodings (corresp.)," *IEEE transactions on information theory*, vol. 12, no. 3, pp. 399–401, 1966.

[21] R. Rice and J. Plaunt, "Adaptive variable-length coding for efficient compression of spacecraft television data," *IEEE Transactions on Communication Technology*, vol. 19, no. 6, pp. 889–897, December 1971.

[22] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[23] *Cortex-A9 Revision r3p0 Technical Reference Manual*, ARM Limited, July 2011, issue G.

[24] Xilinx, *Integrated Logic Analyzer*, https://www.xilinx.com/products/intellectual-property/ila.html.

[25] *DesignSart – ARM Developer*, ARM Limited, 2019, https://developer.arm.com/ip-products/designstart.

[26] ARM Limited, *ARMv8-M Architecture Reference Manual*, 2017, issue A.e.

[27] B. Wijnen, E. J. Hunt, G. C. Anzalone, and J. M. Pearce, "Open-source syringe pump library," *PloS one*, vol. 9, no. 9, p. e107216, 2014.

[28] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapotre, G. Gogniat, and A. K. Biswas, "A novel lightweight hardware-assisted static instrumentation approach for arm soc using debug components," in *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2018, pp. 92–97.

[29] M. N. Allah, G. Hiet, M. A. Wahab, P. Cotret, G. Gogniat, and V. Lapotre, "Hardblare: a hardware-assisted approach for dynamic information flow tracking," 2016.